VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Vizualizace proteinů

Protein Visualization

2014 Bc. Michal Chromec

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Michal Chromec**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Vizualizace proteinů
Protein Visualization

Zásady pro vypracování:

Cílem práce je navrhnout a implementovat vizualizaci molekul proteinů. Důraz je kladen na zobrazení primární, sekundární a terciární struktury proteinů s využitím OpenGL.
Body zadání:
1. Stručná charakteristika struktury proteinů a jejich geometrické reprezentace.
2. Přehled existujících nástrojů pro vizualizaci molekul.
3. Návrh a implementace vlastního řešení.
4. Výkonnostní testy na existující kolekcích proteinů.

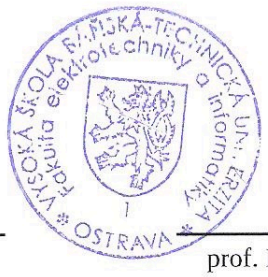
Seznam doporučené odborné literatury:

[1] David Wolff: OpenGL 4.0 Shading Language Cookbook, ISBN: 9781849514767, Packt Publishing Ltd, 2011

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Petr Gajdoš, Ph.D.**

Datum zadání: 16.11.2012
Datum odevzdání: 07.05.2014



doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 29.4.2014 Michal Chramec

Poděkování

Děkují svému vedoucímu diplomové práce Ing. Petrovi Gajdošovi, Ph.D. za odborné vedení a pomoc při zpracování této práce.

INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

Poděkování

Tato práce byla vypracována s podporou projektu Bio-inspirované metody: věda, vzdělávání a transfer znalostí, reg. č. CZ.1.07/2.3.00/20.0073 podpořeného Operačním programem Vzdělávání pro konkurenceschopnost, financovaného ze strukturálních fondů EU a státního rozpočtu ČR.

Abstrakt

Tato práce se zabývá implementaci aplikace k vizualizaci molekul proteinu. V práci jsou popsané základní informace o proteinech a grafické reprezentace molekul. Dále je popsaný PDB soubor, ve kterém jsou uloženy informace o molekule proteinu, hlavně 3D souřadnice jednotlivých atomů. Krátce jsou představeny některé aplikace používané k vizualizaci molekul proteinů. V další části práce je popsaná implementace vlastní aplikace k vizualizaci - detailněji jsou popsané techniky vykreslování jednotlivých reprezentací molekul s pomocí OpenGL a GLSL. V další části popisuji jednoduchý simulátor molekul i s jeho implementací. Poslední část popisuje měření výkonu vytvořené aplikace.

Klíčová slova

Vizualizace proteinů, OpenGL, simulátor molekul

Abstract

This thesis deals with the implementation of the application for the visualization of proteins. There are described the basic information about proteins and graphical representations of molecules. It is described PDB file, which store information about the protein molecule, mainly the 3D coordinates of individual atoms. Briefly present some application used to visualize protein molecules. The next part describe the implementation of own application for visualization - are described in more detail rendering technique individual representation of molecules using OpenGL and GLSL. The next section describes a simple simulator molecules and its implementation. The last section describes the measurement of the performance of the created application.

Keywords

Protein visualization, OpenGL, molecule simulation

Seznam použitých symbolů a zkratek

C++ - označení programovacího jazyku

GLSL - OpenGL Shading Language (programovací jazyk pro shadery)

GPU - Graphic Procesor Unit (grafická karta)

PC - Personal Computer (osobní počítač)

PDB - Protein Data Bank (data banka proteinů)

RAM - Random Access Memory (paměť počítače)

VA - Vertex Array (vertexová pole)

Obsah

1 Úvod ... 1
2 Popis proteinů a jejich geometrická reprezentace ... 2
2.1 Proteiny ... 2
2.1.1 Aminokyselina ... 2
2.1.2 Struktura proteinu ... 2
2.2 Vizualizace molekul proteinů ... 4
2.3 Datový soubor s popisem molekuly ... 5
3 Nástroje k vizualizaci molekul ... 8
3.1 QuteMol ... 8
3.2 PyMol ... 9
3.3 Jmol ... 10
3.4 RasMol ... 11
3.5 UGENE ... 12
4 Návrh a implementace vlastního řešení ... 13
4.1 Knihovna pro práci s molekulami proteinů ... 14
4.2 GUI aplikace ... 17
4.2.1 Okno hlavní aplikace ... 17
4.2.2 Vizualizační okno ... 17
4.2.3 Okno s popisem molekuly ... 17
4.3 Ovládání scény ... 21
4.3.1 Společná kamera ... 21
4.4 Vykreslování ... 23
4.4.1 OpenGL ... 23
4.4.2 Hlavní vykreslovací metoda ... 23
4.4.3 Shadery ... 25
4.4.3.1 Vykreslení koule jako sprite ... 25
4.4.3.2 Vykreslení válce jako sprite ... 27
4.4.4 Vykreslení různých reprezentací molekuly ... 31
4.4.5 Reprezentace založena na vykreslování čar ... 31
4.4.6 Reprezentace založené na vykreslování koulí a válců ... 33
4.4.7 Reprezentace založené na vykreslování křivky ... 34
4.5 Spuštění vytvořené aplikace ... 43
5 Simulátor molekul ... 44
5.1 Rozsah vstupních dat ... 44

5.2 Princip funkce 45
5.2.1 Pravidla chování atomů 45
5.3 Aktuální stav programu 46
5.3.1 Okno simulátoru 47
5.3.2 Vlákno simulace 47
6 Testování aplikace 49
6.1 Rychlost vykreslování 49
6.2 Paměťová náročnost 51
6.3 Testy simulátoru 53
7 Závěr 54
Literatura 55
Seznam příloh 57

1 Úvod

Cílem této práce bylo vytvořit aplikaci k vizualizaci molekul. Vizualizace by měla být založena na OpenGL. Jednou ze schopností aplikace mělo být vizuální porovnávání dvou molekul proteinů. Tedy výsledkem by mělo být, že uživatel zkoumající dvě molekuly nemusí otáčet nebo posouvat každou molekulou zvlášť, ale může použít transformaci na oběma molekuly najednou. Dalším cílem bylo umožnění zvýraznění části molekuly proteinu, tedy aplikace by měl umožnit uživateli vybrat část proteinového řetězce a vybraná část by se projevila zvýrazněním vybrané části přímo ve vizualizaci.

Text práce je rozdělen do následujících kapitol. Ve 2. kapitole jsou uvedeny základní informace o proteinech, také jsou popsány některé grafické reprezentace molekul, které jsou doplněny ukázkou reprezentací používaných aplikací PyMol. Dále je v této kapitole uveden zdroj odkud lze získat zdrojový soubor popisující molekulu proteinu se souřadnicemi jednotlivých atomů a také jsou uvedeny souborové formáty z nichž je jeden vybrán a detailněji popsaný. Ve 3. kapitole jsou popsané některé aplikace věnující se vizualizaci molekul proteinů, jedná se především o popis ovládaní, hlavních funkcí a grafiky. Ve 4. kapitole je popsaná implementaci vlastního řešení. Podrobněji jsou uvedeny použité datové struktury a metoda k načtená dat molekuly ze zdrojového souboru, popsané je GUI aplikace, které je implementována s využitím Qt Frameworku, dále se popit týká ovládání scény. Hlavní část implementace je o samotném vykreslování, které je řešeno s využitím OpenGL a shaderu napsaných v GLSL, jsou popsané hlavní části vykreslování a také postupy vykreslováni různých reprezentací molekul. Celá aplikace je napsaná v programovacím jazyce C++. V 5. kapitole je předveden jednoduchý simulátor molekul, který byl vytvořen jako rozšíření diplomové práce, je zde uveden princip na kterém funguje a také popis jeho implementace. Předposlední 6. kapitola se zabývá měřením výkonu výsledné aplikace. Měření se týká rychlosti vykreslování a paměťové náročnosti aplikace v různých režimech vykreslování, pro různě velké molekuly proteinů. Také byl proveden test simulátoru - test se týkal počtu iterací provedených za jednotku času. V poslední části (7. kapitola) je uvedeno závěrečné zhodnocení dosažených výsledků s možnostmi dalšího vylepšení aplikace.

2 Popis proteinů a jejich geometrická reprezentace

V této kapitole budou podrobněji představeny proteiny, jejich grafické reprezentace a také datový soubor s popisem struktury proteinu. V první části jsou uvedeny základní informace o proteinech, co to vlastně protein je a z jakých částí se skládá. Druhá část se zaměřuje na grafickou reprezentaci molekul proteinů, jsou v ní uvedeny některé grafické reprezentace s popisem a ukázkou v podobě obrázku. Třetí část popisuje odkud a v jakém datovém souboru můžeme popis molekuly proteinu získat, také je v teto části jeden z datových souborů detailněji popsaný.

2.1 Proteiny

Proteiny (bílkoviny, biopolymery), jejichž kostru tvoří polypeptidový řetězec, obsahující obvykle 100 – 2 000 aminokyselinových zbytků. Pořadí aminokyselin v polypeptidovém řetězci je pro každou bílkovinu jedinečné a geneticky dané. Drobné změny primární struktury (vyvolané mutacemi genu) mohou vést k podstatné změně vlastností dané bílkoviny; této skutečnosti využívá proteinové inženýrství, které se cílenými mutacemi snaží změnit (vylepšit) vlastnosti jednotlivých proteinů. V buňce se vyskytuje několik set až tisíc různých bílkovin, které zajišťují její základní funkce a liší se jak chemickou stavbou (především pořadím aminokyselin v peptidovém řetězci - tzv. primární strukturou), tak prostorovým uspořádáním. [1]

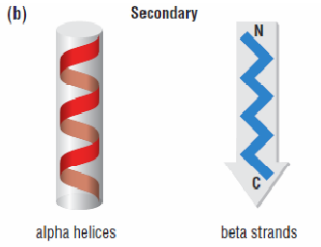
2.1.1 Aminokyselina

Aminokyselina je základním stavebním prvkem proteinového řetězce. Existuje 20 různých druhů. Každá aminokyselina je složena ze dvou částí. První část tvořená skupinou NH_2, alfa uhlíkem C a karboxylovou skupinou COOH, je pro všechny aminokyseliny stejná. Druhá část - postranní řetězce (side chain) odlišuje různé aminokyseliny od sebe a liší se chemickým složením, vychází ze základní části aminokyseliny - je napojený na alfa uhlík. [2]

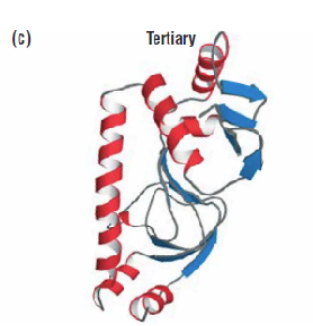
2.1.2 Struktura proteinu

Strukturu proteinu lze reprezentovat čtyřmi úrovněmi. Primární struktura je tvořena sekvencí různých aminokyselin v proteinu. Sekvence aminokyselin, také určují jak se protein bude skládat do struktur vyšších úrovní. Sekundární struktura polypeptidového řetězce může být buď alfa helix nebo beta strand, je dána pravidelnou vodíkovou vazbou, která vzniká interakcí mezi N-H a C=O skupinami v hlavním řetězci (backbone). Pravidelné struktury (helix, strand) části řetězce jsou prokládaný nepravidelnými strukturami označovanými jako loop nebo coil. Terciární struktura představuje celý řetězec složený do kompaktní struktury, tvořený prvky sekundární struktury (helix, strand) a spoji mezi nimi. Kvartérní struktura představuje protein, který je složený z více než jednoho polypeptidového řetězce. [2] [3]

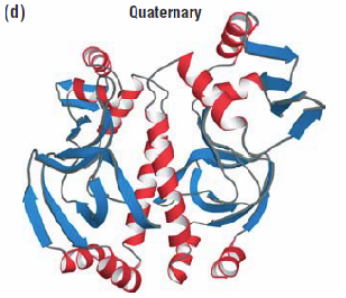
Obrázek 2-1: Primární struktura [2]



Obrázek 2-2: Sekundární struktura [2]



Obrázek 2-3: Terciární struktura [2]



Obrázek 2-4: Kvartérní struktura [2]

Alfa šroubovice (Alpha helix)

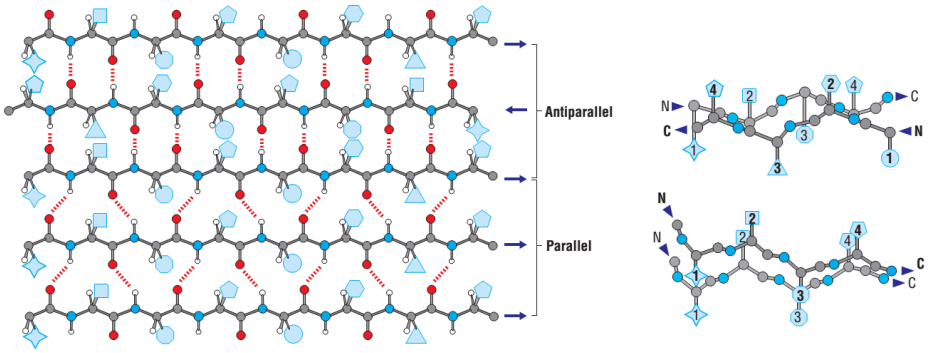
Alpha šroubovice je nejčastějším prvkem sekundární struktury ve složeném polypeptidovém řetězci. Struktura helixu je stabilizována vodíkovou vazbou mezi C=O a N-H skupinami. V alpha helixu, atom z C=O skupiny každého residua (n) přijímá vodíkovou vazbu z NH skupiny čtvrtého následujícího residua v pořadí (n+4) v sekvenci. Výsledkem je cylindrická struktura, ve které je zeď válce tvořena vodíkovými vazbami hlavního řetězce (backbone) a vnější část je posetá postraním řetězcem (side chain). Vyčnívající postranní řetězce určují interakce alpha helixu s ostatními částmi složeného proteinového řetězce a také s jinými proteiny. [2]

Dále existují i jiné varianty helix struktur [4], jedná se o:

- 3-10 helix - vodíková vazba je tvořena mezi residui vzdálenými od sebe o 3 pozice v řetězci
- pi-helix - vodíková vazba je tvořena mezi residui vzdálenými od sebe o 5 pozice v řetězci

Skládaný list (Beta sheet)

Beta skládaný list je tvořený vodíkovými vazbami mezi C=O a N-H skupinami residui, které jsou v od sebe vzdálené v proteinovém řetězci. Skládaný list je tvořený dvěma nebo více vlákny (strands), které mohou být daleko od sebe v proteinovém řetězci, vlákna jsou uspořádaná vedle sebe s vodíkovými vazbami mezi sebou. Vlákna mohou běžet ve stejném směru (parallel beta sheet) nebo proti sobě (antiparallel), možné jsou také smíšené listy (mixed sheets) s paralelními a antiparalelními vlákny. [2]



Obrázek 2-5:Skládaný list [2]

2.2 Vizualizace molekul proteinů

Při vizualizaci molekul se vychází z pozic jednotlivých atomů a používají se různé reprezentace při vykreslování molekul proteinů. V této kapitole budou uvedeny základní reprezentace, které se používají k vizualizaci molekul proteinů a jsou součásti většiny vizualizačních aplikací. Pro lepší představu o reprezentacích je uveden obrázek 2-6 s ukázkou reprezentací používaných aplikací PyMol. Popis vykreslení jednotlivých reprezentací je podle dokumentace aplikace VMD [5]. Nejčastější způsoby vizualizace jsou tedy následující:

Lines (wireframe) - vykreslují se jen vazby mezi atomy jako čára mezi pozicemi dvou atomů

Sticks (bonds, Licorice) - vazby jsou vykresleny jako válce, okraj válce je zakončen polokouli, válec i koule má stejný poloměr

Balls and Stick - atomy jsou vykresleny jako koule a vazby mezi nimi jako válce

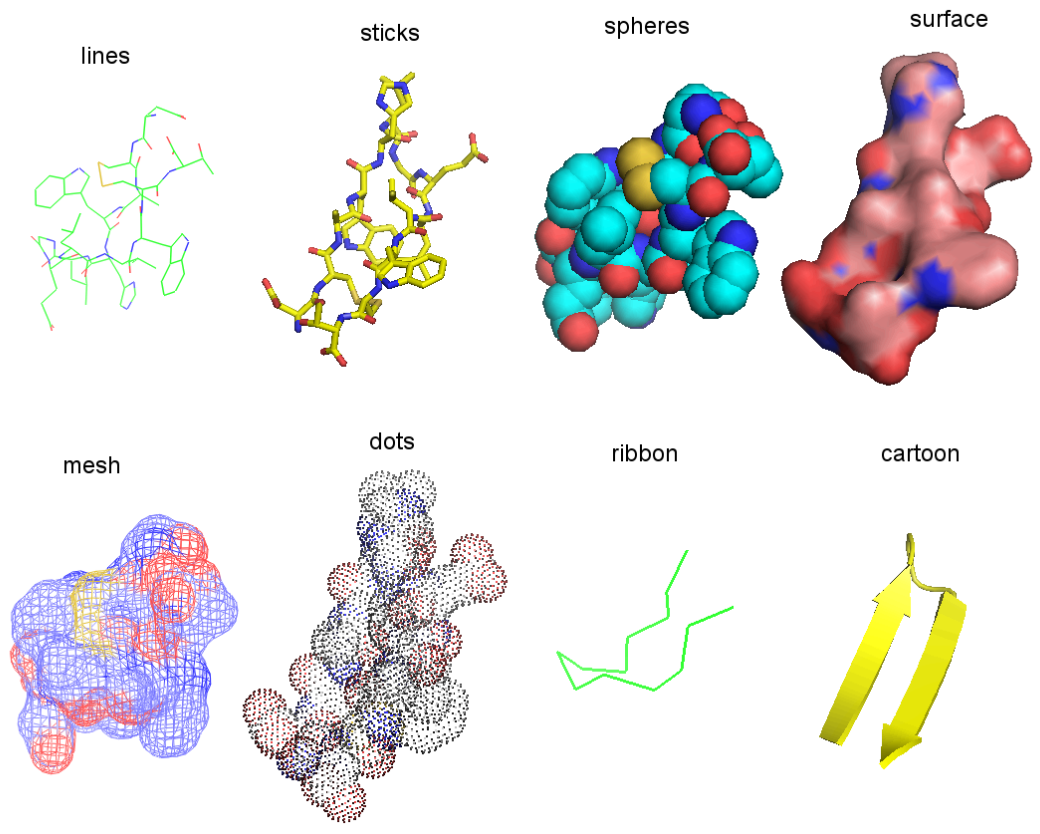
CPK (space-filling) - atomy jsou vykresleny jakou koule s Van der Waalsovýmv poloměrem

Trace - vykreslují se válce spojující alfa uhlíky po sobě jdoucích aminokyselin

Tube - vykresluje se křivka(potrubí) přes atomy alfa uhlíků po sobě jdoucích aminokyselin

Cartoon - zjednodušená reprezentace proteinu založená na vykreslování prvků sekundárních struktur, šroubovice (helix) je vykreslená jako pás zatočený do spirály, vlákno skládaného listu (strand) je vykreslený jako pás zakončený šipkou, zbytek řetězce je vykreslen jako křivka

Surface - vykresluje se povrch molekuly



Obrázek 2-6 Ukázka reprezentací z aplikace PyMol

2.3 Datový soubor s popisem molekuly

Pro vizualizaci molekuly proteinu je potřeba získat soubor s popisem struktury molekuly a popisem pozic jednotlivých atomů. Zdrojem takových to dat může být např.: databanka proteinů (Protein Data Bank - PDB [14]). Archív PDB obsahuje informace o experimentálně získaných strukturách proteinů, nukleových kyselin a složitějších sestav.

PDB archív poskytuje data o molekulách proteinů ve třech souborových formátech:

- PDB - obsah souboru je formátován do řádků, kde každý řádek začíná názvem záznamu, tento záznam přesně definuje do bude za ním následovat.
- mmCIF - obsah souboru je založený na slovníkové struktuře, tedy každá položka je dána dvojici - popisovaného parametru a jeho hodnoty
- PDBML - obsah souboru je v XML formátu

Z souborových formátu byl vybrán PDB formát jako výchozí soubor k načtení struktury proteinu a souřadnic jeho atomů. Důvodem výběru byla především jednoduchost tohoto formátu a také přehledná specifikace[13]. V následující části bude formát PDB souboru detailněji popsaný.

Popis PDB souboru s ukázkami záznamů

PDB soubor [13] je textový soubor, ve kterém jsou uloženy informace o struktuře proteinu. Hlavní části souboru jsou záznamy se souřadnicemi atomů, ze kterých je protein složený. Dále tento soubor poskytuje informace o sekundárních strukturách proteinu (helix, sheet).

Záznamy v souboru jsou zapsány do řádků o délce 80 znaků. Každý řádek začíná názvem (typem) záznamu (record name) a podle typu záznamu je složený z jednoho nebo více řádků. Každý řádek je tedy specifický názvem záznamu a případnými argumenty uvedenými za ním. Pořadí záznamů v PDB souboru je dáno specifikací a zde uvádím seznam hlavních názvů záznamu s příkladem a popisem, tak jak za sebou následují:

HEADER - hlavička souboru, obsahuje označení proteinu
HELIX - záznam o pozici helixu v řetězci proteinu
SHEET - záznam o pozici strandu v řetězci proteinu
MODEL - uvádí číslo modelu molekuly v případě, že je v souboru uloženo více modelů
ATOM - informace o atomu
ENDMODEL - ukončení popisu aktuálního modelu
END - konec PDB souboru

Pro každý záznam obsahuje specifikace seznam parametrů, které za názvem záznamu následují. Jejich popis s přesně danou pozicí v řádku je uveden ve specifikaci souboru[13]. V následujících částech jsou uvedeny ukázky záznamů HELIX, STRAND a ATOM s popisem jejich parametrů.

Ukázka záznamu HELIX:

```
HELIX    1   1 PRO P  100  THR P  104  5                                    5
HELIX    2   2 ALA P  153  THR P  155  5                                    3
HELIX    3   3 GLY P  175  PHE P  177  5                                    3
HELIX    4   4 HIS P  196  ILE P  198  5                                    3
```

V ukázce jsou popisovány čtyři struktury helix, parametry každého řádku jsou přesně dané. Zde je uveden podrobný popis prvního řádku:

- HELIX - název záznamu
- 1 - sériové číslo helixu
- 1 - identifikátor helixu
- PRO - jméno počátečního residua
- P - identifikátor řetězce, ve kterém je tento helix
- 100 - sekvenční číslo počátečního residua
- THR - jméno koncového residua
- P - identifikátor řetězce, ve kterém je tento helix
- 104 - sekvenční číslo koncového residua
- 5 - třída helixu
- 5 - délka helixu

Ukázka záznamu SHEET:

```
SHEET    1   B 3 VAL A 430  ASN A 432  0
SHEET    2   B 3 SER A 437  PHE A 442 -1  O  SER A 437   N  ASN A 432
SHEET    3   B 3 ARG A 445  GLN A 448 -1  N  ARG A 445   O  PHE A 442
```

Jedná se o ukázku popisu skládaného listu se třemi vlákny. Podrobný popis druhého řádku je následující:

- SHEET - název záznamu
- 2 - číslo vlákna (strand), které tvoří skládaný list (sheet)
- B - identifikátor skládaného listu (sheet)
- 3 - počet vláken ve skládaném listě
- SER - název počátečního residua
- A - identifikátor řetězce, ve kterém se nachází počáteční reziduum vlákna
- 437 - číslo počátečního residua
- PHE - název koncového residua
- A - identifikátor řetězce, ve kterém se nachází koncové reziduum vlákna
- 442 - číslo koncového residua
- -1 - určuje směr vlákna oproti předchozímu vláknu (0-první vlákno, 1-paralelní, -1-antiparalelní)

O SER A 437 N ASN A 432 - popisuje umístění vodíkové vazby mezi aktuální a předchozím vláknem (u prvního vlákna není uvedena)

Ukázka záznamů ATOM:

```
ATOM      1  N   SER P  71     -47.333   0.941   8.834  1.00 52.56           N
ATOM      2  CA  SER P  71     -45.849   0.731   8.796  1.00 53.56           C
ATOM      3  C   SER P  71     -45.191   1.608   7.714  1.00 51.61           C
```

```
ATOM      4  O   SER P  71     -45.455   2.818   7.648  1.00 54.49           O
ATOM      5  CB  SER P  71     -45.511  -0.764   8.600  1.00 55.68           C
ATOM      6  OG  SER P  71     -46.116  -1.305   7.434  1.00 58.53           O
ATOM      7  N   GLY P  72     -44.347   1.018   6.868  1.00 46.18           N
ATOM      8  CA  GLY P  72     -43.702   1.805   5.836  1.00 38.59           C
ATOM      9  C   GLY P  72     -43.533   1.109   4.498  1.00 34.81           C
ATOM     10  O   GLY P  72     -44.500   0.739   3.827  1.00 32.75           O
ATOM     11  N   PHE P  73     -42.276   0.924   4.128  1.00 29.72           N
```

Ukázka popisuje 11 záznamů ATOM, z prvních šesti záznamů lze vyčíst, že patří jednomu residuu SER s identifikátorem 71. Dále lze vyčíst, že všechny atomy patří do stejného řetězce s označením P. Podrobný popis prvního řádku je následující:

- ATOM - název záznamu řádku a znamená to, že budou následovat údaje o atomu
- 1 - sériové číslo atomu
- N - název atomu
- SER - název rezidua (aminokyseliny), jehož součástí je tento atom
- P - identifikátor řetězce (sekvence), který je tvořený rezidui
- 71 - sekvenční číslo rezidua
- -47.333 0.941 8.834 - souřadnice atomu (X Y Z)
- 1.00 52.56 - obsazenost (occupancy) a teplotní faktor
- N - označení atomu

3 Nástroje k vizualizaci molekul

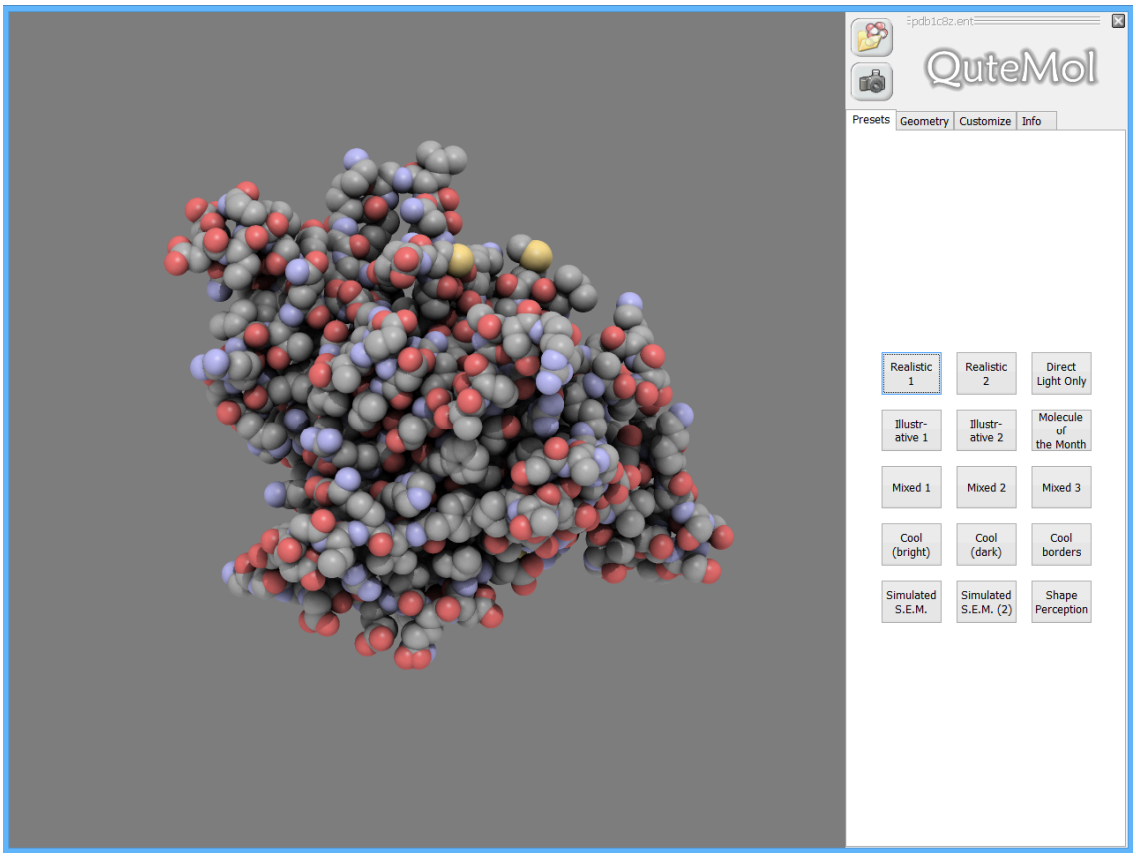
V této části jsou popsány některé aplikace používané k vizualizaci molekul proteinu. Popis se týká hlavně základního ovládání aplikace a vizualizaci.

3.1 QuteMol

Jednoduchá vizualizační aplikace, která umožňuje načíst molekulu z PDB souboru a vykreslit v režimech Balls-and-sticks, Licorice a Space-fill. Vykreslování je založeno na technikách uvedených v [6]. V tomto dokumentu jsou detailněji popsané techniky vizualizace molekul se zaměřením na molekuly větších rozměrů a vykreslování ve vysoké kvalitě. Aplikace je v podstatě ukázka implementace těchto technik.

Aplikace je tvořena vizualizačním oknem a panelem s nastavením a tlačítky pro načtení PDB souboru a pro vytvoření obrázku scény. Scénu lze ovládat (rotace, zoom) myší ve vizualizačním okně.

Dále aplikace umožňuje měnit parametry efektů, osvětlení nebo barvu pozadí, je zde také možnost použití přednastavených efektů. Pro režimy vykreslení molekuly umožňuje nastavit poloměry atomů a vazeb.[6] [7]



Obrázek 3-1: Aplikace QuteMol

3.2 PyMol

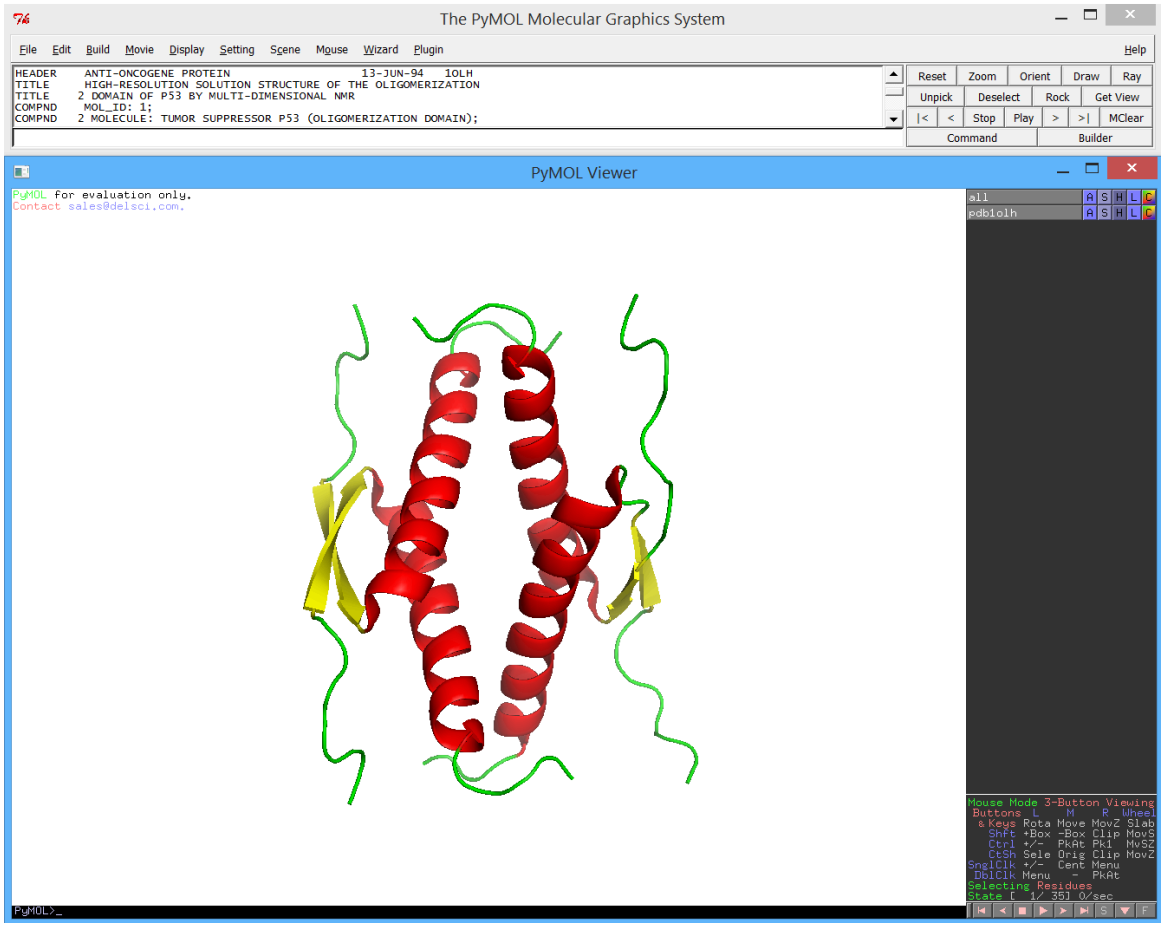
PyMol je jedna z populárních aplikací, jedná se o komplexní nástroj pro vykreslování a animování 3D molekulárních struktur.

Hlavní část aplikace tvoří okno, ve kterém se vykresluje 3D struktura proteinu. V tomto okně je možné pomocí myši scénu ovládat - rotace scény, posun, přiblížení/oddálení. Dále jsou podporovány kombinace klávesnice a myší, kterými lze scénu ovládat nebo také slouží k výběru části molekul. Aplikace podporuje řadu typu zobrazení: pro vizualizaci vazeb (lines, sticks) pro prezentaci sekundární struktury proteinu (ribbon, cartoon), dále reprezentace atomů jako koule a dále objemové pohledy (mesh, surface), popis aplikace uvádí 20 různých způsobů reprezentace.

Další funkcí aplikace je, že vytvořenou scénu lze vykreslit pomocí techniky ray-tracing, čímž docílíme ještě pěknějšího vzhledu scény-u zakulacených objektů nejsou vidět hrany, zobrazují se stíny, lepší osvětlení, nevýhodou této techniky je náročnost - neběží v reálném čase.

Dalším nástrojem této aplikace je příkazový řádek, pomocí něhož lze vizualizaci ovládat a s pomocí různých příkazů a parametrů měnit, v popisu aplikace [8] se uvádí více než 600 nastavení. Dále zvládá přes 30 různých souborových formátů.

Jedná se o open-source aplikaci, k dispozici jsou zdarma ke stažení zdrojové soubory poslední verze aplikace, které si musí uživatel sám sestavit. Sestavené řešení je placené-podpora vývoje aplikace. [9]



Obrázek 3-2: Aplikace PyMol

3.3 Jmol

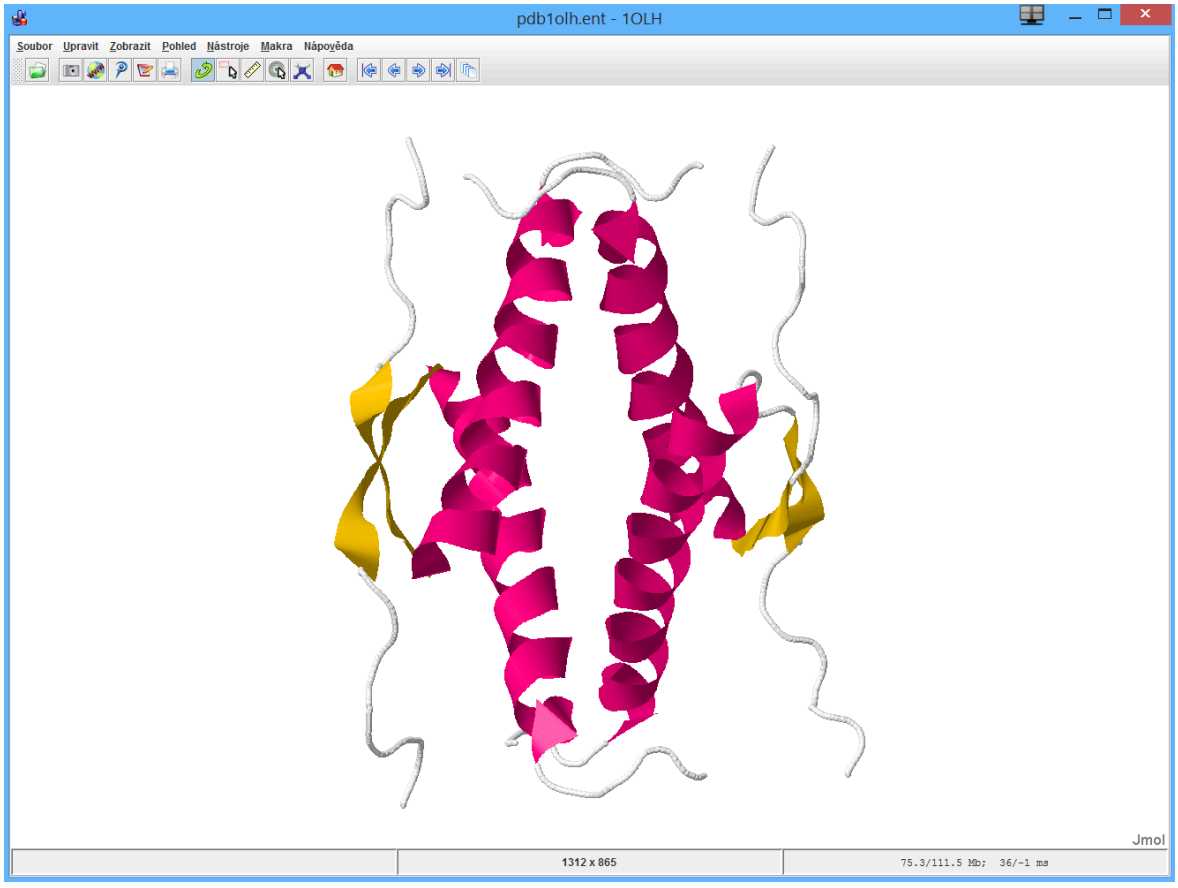
Jedná se o open-source vizualizační aplikaci založenou na Javě [10]. Hlavní části aplikace je 3D vizualizační okno, ve kterém se vykresluje model molekuly proteinu. Ovládání scény je možné pomocí myši a kombinace klávesnice a myši. Další možnosti aplikace lze měnit přímo v GUI okně aplikace pomocí tlačítek, přes menu, nebo vyvoláním menu ve vizualizačním okně přes pravé tlačítko myši. Tuto aplikaci lze použít ve formě webové aplikace (JmolApplet) a použít ji na webových stránkách.

Zajímavostí této aplikace je, že používá vlastní vykreslovací engine, který je kompletně napsaný v javě. Vykreslování tedy není řešeno pomocí hardware (např.: OpenGL), ale softwarově.

Aplikace umožňuje vytvoření obrázku aktuální scény, dále je zde možnost vykreslit scénu pomocí aplikace POV-ray - jmol vygeneruje soubor pro tuto aplikaci, scéna je poté vykreslena pomocí techniky ray-tracing.

Schéma vykreslení molekuly, ke kterým se lze dostat přes menu je šest typů:

- Kalotový model CPK
- Kuličky a tyčinky
- Tyčinkový model
- Drátový model
- Skica (cartoon)
- Sledování (trace)



Obrázek 3-3: Aplikace Jmol

3.4 RasMol

RasMol je důležitý vědecký nástroj k vizualizaci molekul, který vytvořil Roger Sayle v roce 1992. RasMol je používán tisíci lidmi po celém světě ke zobrazování makromolekul a přípravě obrázku k publikování. Umožňuje vizualizaci proteinu, nukleových kyselin a malých molekul. Je určen k zobrazování, vzdělávání a generování obrázku publikační kvality. Aplikace může běžet na různých operačních systémech (Microsoft Windows, Apple Macintosh, UNIX). [11]

Aplikace je tvořena hlavním 3D vizualizačním oknem s menu nabídkou a oknem s příkazovou řádkou. Přes menu aplikace lze zvolit následující režimy vykreslování molekuly: wireframe, backbone, sticks, spacefill, balls and sticks, ribbons, strands, cartoons, molecular surface. Dále lze přes menu měnit nastavení aplikace, měnit barevné režimy molekuly a exportovat scénu např.: jako obrázek, POV-ray.

Aplikace podporuje řadu vstupních souborů se souřadnicemi molekul, např.: PDB, Mol2, MDL, CIF, mmCIF.

Vykreslenou molekulu můžeme otáčet, posouvat, přiblížit/oddálit a části molekuly můžeme odstranit ořezovou rovinou. Scénu můžeme ovládat myši, klávesnici, posuvníky okna, příkazovou řádkou nebo přes hlavní menu.

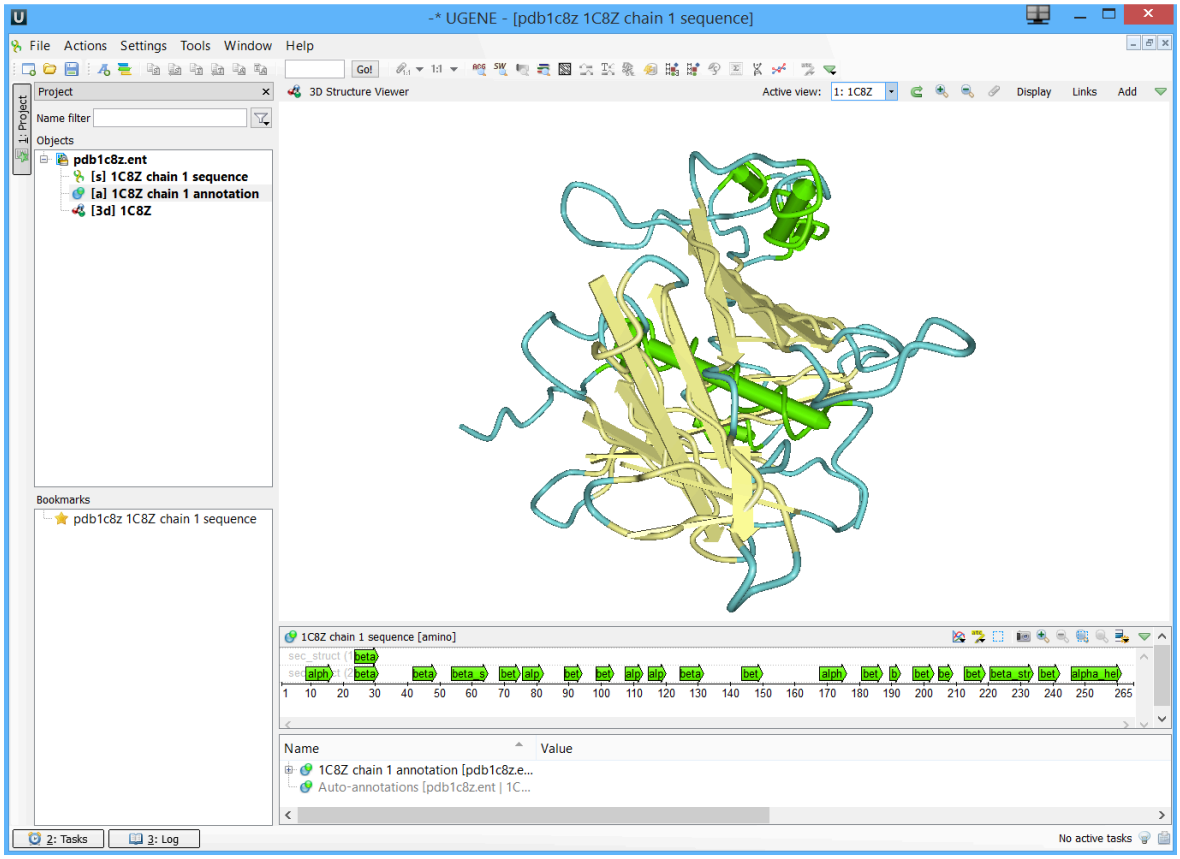
Obrázek 3-4: Aplikace RasMol

3.5 UGENE

Unipro UGENE je multiplatformní open-source software pro správu, analýzu a vizualizaci dat v bioinformatice. [12]

Hlavní část aplikace je tvořena 3D vizualizačním oknem s vykreslenou molekulou. Na okraji je panel s informacemi o aktuálním projektu (název souboru, popis proteinu a jeho řetězců). Ve spodní části aplikace jsou dva panely s detailnějším popisem řetězců a celé molekuly. Řetězec je zobrazen jako číslovaná řada residuí nad nimiž je značkou vyobrazena i sekundární struktura (strand, helix). Takto zobrazený řetězec lze i přiblížit a zobrazí se i označení jednotlivých aminokyselin v řetězci. Další panel zobrazuje molekulu jako stromovou strukturu, z které lze vyčíst informace o jednotlivých řetězcích nebo prvcích sekundární struktury. Panely jsou propojeny s vizualizačním oknem, ve kterém se zvýrazní část vykreslované molekuly na základě výběru z panelů.

Vykreslované reprezentace molekul jsou: ball and stick, space fill, tubses, worms. Dále umožňuje vykreslené povrchu molekuly třemi způsoby. Další možnosti je barevné odlišení části molekul podle: typů prvku, řetězce, sekundární struktury nebo vše jednou barvou.



Obrázek 3-5: Aplikace UGENE

4 Návrh a implementace vlastního řešení

Tato kapitola popisuje návrh a implementaci vlastní aplikace. Návrh implementace je rozdělen do čtyř bodů. K těmto bodům jsou poté vytvořené samotné kapitoly zabývající se jednotlivými body. Hlavní body jsou následující:

- V první fázi bylo nutné vytvořit datové struktury popisující molekuly proteinu a metodu, která by tyto struktury naplnila daty ze zdrojového souboru. Pro tuto část byla implementována samostatná knihovna.

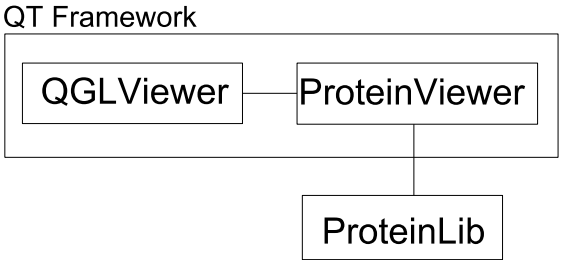
- Druhá fáze se týkala grafického uživatelského rozhraní, které bylo nutné k ovládání implementované aplikace.

- V další fázi bylo nutné promyslet implementaci ovládání vykreslované molekuly, aby byla umožněn posun či rotace vykreslovaného objektu. S tímto bodem také souvisí jeden z cílů funkčnosti aplikace a to aby byla možnost sdílení transformací mezi okny - uživatel tak bude moct posouvat nebo otáčet dvěma molekulami najednou (nap.: při porovnávaní).

- Poslední část se týká samotného vykreslování. Samozřejmostí bylo použití OpenGL k vykreslování. Bylo tedy nutné zvolit nejefektivnější způsob, aby bylo vykreslování plynulé. Dále bylo nutné promyslet, které reprezentace se budou jak vykreslovat a také zohlednit při vykreslování možnost zvýraznění části molekuly na základě výběru uživatele.

Aplikace byla vyvíjená v programovacím jazyce C++ ve vývojovém prostředí Microsoft Visual Studio, ve kterém je rozdělena do tří samostatných projektů (obrázek 4-1):

- ProteinViewer - spustitelná aplikace k vizualizaci proteinů
- ProteinLib - knihovna, obsahuje datové struktury a metody pro načtení proteinu ze souboru
- QGLViewer - knihovna pro manipulaci s 3D scénou



Obrázek 4-1: Projekty (části) aplikace

Jednotlivé projekty jsou uvedeny na následujícím obrázku, QGLViewer a ProteinViewer jsou založeny na QT Frameworku. ProteinLib je pouze knihovna, kterou používá ProteinViewer.

4.1 Knihovna pro práci s molekulami proteinů

Tato knihovna byla vytvořena pro definici datové struktury (třídy) molekuly proteinu, metody pro načtení struktury proteinu ze zdrojového PDB souboru, definice konstant parametrů atomů a definice barev pro různé části molekuly. V první části jsou uvedeny základní struktury s popisem. Následující část popisuje načtení molekuly proteinu ze souboru PDB. Další části se týkají poloměrů atomů a barevných schémat, používaných při vykreslování.

Datové struktury

Pro načtení molekuly proteinu a následnou práci s ní bylo třeba nejprve vytvořit struktury (třídy) představující jednotlivé části molekuly, tyto části odpovídají reálné struktuře proteinu. Pro jednotlivé části molekuly byly vytvořeny tedy následující třídy:

Atom - třídá popisující jeden atom molekuly, je v ní uloženo seriové číslo atomu uvedené v PDB souboru (aId), název atomu (name), název residua (resName), oznaccení řetězce (chainId), číslo residua (resId), souřadnice atomu (coord) a symbol prvku (sybmol)

Residue - třída popisující aminokyselinu v řetězci - residuum, obsahuje pole atomů tvořící toto residuum

Chain - třída popisující řetězec proteinu, obsauhje pole rezidují, ze ktercýh je složený, dále obsahuje pole sekundárních struktur, které se v řetězci nacházejí (helix, strand)

Molecule - třída popisující celou molekulu proteinu, obsahuje pole řetězců, ze ktercýh se molekula skládá, dále obsahuje pole všech residuí, atomů a vazeb pro jednodušší procházení, když je potřeba procházet všechny prvky najednou, dále obsahuje název souboru, ze kterého se načetli data a identifikátor proteinu načtený z hlavičky souboru (idCode)

Strand - třída popisující jeden pás skládaného listu (sheet), uchovává informace o pozici vlákna v molekule proteinu - tedy id řetězce, číslo počátečního a koncového residua a identifikátor sheetu

Helix - třída popisující sekundární strukturu helix, uchovává informace o pozici helixu v molekule proteinu - id řetězce, číslo počátečního a koncového residua, dále obsahuje informaci o třídě helixu (alfa helix, pi-helix, 3-10 helix)

Načtení dat

Cílem bylo napsat metodu, která by ze vstupního souboru přečetla důležité informace molekule proteinu a vytvořila objekt molekuly podle třída *Molecule* uvedené v předchozí části. Objekt molekuly je pak dále využívána v dalších částech aplikace, kde se podle něj vykresluje molekula proteinu v trojrozměrném prostoru nebo slouží k výpisu informací o molekule (názvy atomů, názvy aminokyselin).

Pro zobrazení proteinu ve 3D je potřeba znát strukturu molekuly-které atomy tvoří residua a dále ze kterých residuí jsou tvořeny řetězce molekuly. Dále je potřeba znát umístění sekundárních struktur v řetězcích proteinu a souřadnice jednotlivých atomů. Data o jednotlivých atomech jsou uvedeny v PDB souboru, pro každý atom je v PDB souboru jeden řádek se záznamem ATOM. Vytvoření objektu molekuly spočívá ve čtení záznamů atomů, z nichž se postupně sestavuje celý objekt

molekuly. Při načítání molekuly je nejprve potřeba prozkoumat PDB soubor a zjistit, zda se v souboru nachází jeden nebo více modelů molekul. To se docílí čtením souboru řádek po řádku a porovnáváním jestli řádek začíná slovem MODEL, v případě, že začíná inkrementuje se proměnná, ve které je uložen aktuální počet nalezených slov MODEL. Po přečtení celého souboru víme, kolik modelů je v PDB souboru obsaženo a tak se můžeme dotázat uživatele, se kterým modelem by chtěl pracovat. Když známe model, se kterým se bude pracovat, stačí opět číst jednotlivé řádky a počítat záznamy MODEL dokud se nedostaneme na požadovanou pozici, od které můžeme začít zpracovávat záznamy ATOM. V případě že je v souboru uložen jen jeden model molekuly, tak slovem MODEL uveden není a mohou se rovnou zpracovávat záznamy ATOM.

Po přečtení všech záznamů ATOM je nutné zpracovat záznamy sekundární struktury (HELIX, SHEET). V těchto záznamech jsou informace o sekundární struktuře a také o jejím umístění v proteinovém řetězci (identifikátor řetězce, počáteční a koncové residuum).

Poloměry atomů

Při některých režimech vykreslování molekul je potřeba znát mezi kterými atomy je kovalentní vazba. Jedná se o režimy wireframe, balls and sticks, sticks, backbone, ve kterých se vykresluje právě i tato vazba. Testování dvojic zda jsou spojeny kovalentní vazbou probíhá na základě výpočtu vzdáleností dvou atomů od sebe a v případě, že vzdálenost mezi nimi je v intervalu:

<0.4; covRadA1 + covRadA2 + 0.56>

tak jsou považovány za spojené kovalentní vazbou [18], parametry covRadA1 a covRadA2 jsou kovalentní poloměry atomů, všechny hodnoty jsou v jednotkách Angstroms.

Testovaní dvojic atomů, zda mezi nimi je kovalentní vazba probíhá na základě struktury molekuly, ve které jsou kovalentními vazbami spojeny atomy jednotlivých residují a následně atomy sousedních residují podílející se na peptidové vazbě. Při hledání kovalentních vazeb je nutné procházet dvojíce atomů v rámci jednoho residua a v případě, že vzdálenost mezi atomy je v uvedeném intervalu považuji se tyto atomy za spojené kovalentní vazbou. Dále se testuje stejným způsobem polypeptidová vazbu, která je mezi sousedními residui, konkrétně mezi atomem dusíku residua a atomem uhlíku předchozího residua.

Další možnosti, kde se může nacházet kovalentní vazba je mezi atomy síry různých řetězců (disulfide bridge [2]), testuji se tedy ještě dvojice atomů síry v rámci celé molekuly zda jejich vzdáleností spadají do intervalu, ve kterém by atomy šlo brát jako spojené kovalentní vazbou.

Pro výpočet maximální vzdáleností atomů, které by dali považovat za spojené kovalentní vazbou, bylo třeba definovat pro každý typ atomu velikost kovalentního poloměru. Kovalentní poloměry jsou definovány podle tabulky zdroje [18].

Současně byly také definovány Van der Waalsovy poloměry atomů, které jsou použity při vykreslování atomů jako koule. K definování poloměrů byla použita tabulka van der Waalsových poloměrů zdroje [18].

Barevná schémata

Pro rozlišení jednotlivých části molekuly proteinu se používají různá barevná schémata. Bylo tedy nutné definovat barevná schémata, která se budou využívat při vykreslování. Schémata se používají na úrovní atomů, aminokyselin, řetězců nebo k rozlišení části molekuly podle prvků sekundárních struktur. Barevná schémata jsou definována podle aplikace RasMol, podobné schémata používá i aplikace Jmol.

Barevné schéma pro rozlišení jednotlivých atomů vychází z populárního spacefill modelu, který vytvořili Corey, Pauling a později vylepšil Kultun. Barevné schéma pro aminokyseliny je vytvořené na základě vlastností aminokyselin - pro polární jsou použity světlejší barvy, pro nepolární barvy tmavší. [19] [20] [21]

4.2 GUI aplikace

Při návrhu aplikace bylo nutné vytvořit tři typy oken (formulářů), které se budou používat k ovládání aplikace. Jedná se o okno hlavní aplikace, které se zobrazí po spuštění aplikace. Dále bylo nutné implementovat vizualizační okno, ve kterém se bude vykreslovat molekula proteinu. Nutné bylo zohlednit, aby šlo zobrazit více vizualizačních oken najednou k porovnání molekul. Třetím typem okna mělo být okno s popisem molekuly. Popisy jednotlivých oken s implementací jsou uvedeny v následujících kapitolách.

4.2.1 Okno hlavní aplikace

Okno hlavní aplikace, je zobrazeno po spuštění aplikace. Implementace tohoto okna je ve třídě *ProteinViewer*, funkce hlavního okna je zajištěna děděním z třídy Qt Frameworku *QMainWindow*. Jako centrální prvek okna je použitý objekt *QMdiArea* - tento objekt představuje oblast, ve které se mohou zobrazovat další okna. Součástí hlavního okna je defaultně i menu, pomocí něj lze otevřít a zobrazit nové okno molekuly proteinu (načtení ze souboru), spustit okno se simulaci (kapitola 5) a nebo celou aplikaci ukončit. V oblasti *QMdiArea* je zobrazeno podokno informačního panelu (*InfoView*) s výpisem informací o aktuálně otevřených proteinech.

4.2.2 Vizualizační okno

Vizualizační okno představuje okno ve kterém se vykresluje molekula proteinu. Implementace tohoto okna je ve třídě *PtoteinView*, funkce okna je zajištěna děděním z třídy *QMainWindow*. Součásti okna je menu, pomocí kterého, lze vybrat různé reprezentace molekuly k vizualizaci a také volit mezi barevnými režimy. Jako centrální prvek je do okna umístěný widget představující vizualizační OpenGL okno - *Viewer*, který je popsaný v následující části.

Viewer

Třída *Viewer* představuje vykreslovací okno OpenGL, vychází z třídy *QGLViewer*, čímž je zajištěná základní funkčnost při ovládání scény myší. Jedná se hlavně o přiblížení, oddálení, rotaci nebo posun ve 3D scéně. Pro vykreslování bylo třeba dopsat následující metody:

- init() - tato metoda se volá jen jednou před vykreslováním a v této metodě by se měli nastavit potřebné parametry OpenGL jako barva pozadí, depth buffer, cull face, inicializace shaderu

- draw() - tato metoda se volá vždycky, když je potřeba překreslit okno, např.: při změně pohledu na scénu, při změně vykreslovacího režimu nebo při změně barevného režimu

4.2.3 Okno s popisem molekuly

Jedním z požadavků na aplikaci bylo vytvoření okno, ve kterém by zobrazovaly informace o aminokyselinách, atomech aktuálně načtených a zobrazovaných proteinů nebo který by umožňovalo načtení a zobrazení vlastních dat. Další funkcí tohoto okna měla být možnost výběrů aminokyselin s promítnutím výběru do vizualizačního okna - zvýraznění vybrané části molekuly. Další funkci tohoto okna je automatická změna zobrazení informací v řádku nebo ve sloupci v závislosti na aktuálním rozměru okna. V následujících částech bude popsaný princip zvýraznění části molekuly na základě výběru a také samotná implementace tohoto okna.

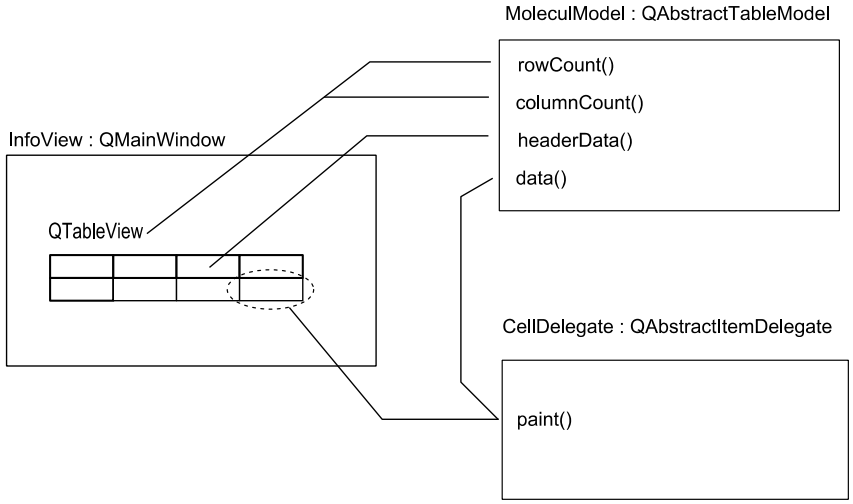
Zvýraznění části molekuly

Při označení některých residují se automaticky zvýrazní i část molekuly v *ProteinView* okně. Když dojde k označení buněk tabulky (*QTableView*) vytvoří se signál *selectionChanged,*tento signál je zachycen ve třídě *MoleculModel* ve slotu *selectionChangedSlot.* Tento signál posílá indexy buněk, u kterých došlo k výběru a index buněk, u kterých byl výběr zrušen. V případě, že je v aplikaci zobrazeno více molekul, obsahuje i tabulka *InfoView* okna více řádků/sloupců popisující jednotlivé molekuly. Ve slotu je tedy nutné indexy rozdělit podle molekul a indexy zaslat *ProteinView* oknům s danými molekulami. *ProteinView* tyto indexy poté zasílají do třídy *Viewer*, kde se podle indexů upraví aktuální seznam části molekuly k zvýraznění a podle něj poté probíhá vykreslení popsané v kapitole 4.4.2.

Popis implementace InfoView okna

Třída *InfoView* dědí z GUI prvku QT frameworku, konkrétně se jedná o *QMainWindow* (klasické okno s menu nabídkou a centrální oblasti pro umístění dalších GUI prvků). Jako centrální GUI prvek je do okna umístěná tabulka (*QTableView*). Tabulka pracuje na základě návrhového vzoru MVC (Model-View-Controller). Qt framework tento návrhový vzor upravil a zkombinoval view a controller do jednoho objektu-view, architektura v Qt frameworku se nazývá Model/View. Tabulka představuje objekt *View*, který slouží k prezentaci dat. Pro zobrazení dat se musí tabulce nastavit objekt-model, ze kterého bude brát data. Model bude podrobněji rozveden v další části textu. Další možnosti tabulky je definovat si vlastní zobrazení buněk skrze delegáta (Delegate).

Princip fungování je znázorněn na obrázku 4-2. Rozměry tabulky se přizpůsobí podle počtu řádků a sloupců získané z modelu (metody *rowCount()* a *columnCount()*). Dále je podstatné vykreslení jednotlivých buněk tabulky, to probíhá tak, že pro danou buňku se volá metoda *paint()* delegáta. Metoda *paint()* provádí samotné vykreslení buňky - potřebná data jako jsou barva a text si metoda zjistí z modelu metodou *data()*.



Obrázek 4-2: InfoView

Dále byla implementována funkce přizpůsobení obsahu podle rozměru okna. Při každé změně velikosti *InfoView* okna se kontrolují rozměry (metoda-událost *resizeEvent* ve třídě *InfoView*) a v případě, že je větší šířka okna, tak se data jedné molekuly vypisují do řádků, v opačném případě, kdy je okno větší na výšku, tak se data vypisují do sloupců. Data poskytuje model, proto je potřeba při změně rozměru okna informovat model - to je zajištěno signálem *changeColumnDir(VERTICAL).*

Model

Model pro tabulku v *InfoView* okně zajišťuje třída *MoleculModel*. Aby mohla být tato třída modelem a zajistit správnou funkci v rámci Model/View architektury z QT frameworku musí dědit z třídy *QAbstractTableModel*. *QAbstractTableModel* poskytuje standardní interface pro model reprezentující data jako dvourozměrné pole (tabulka), při dědění z této třídy bylo nutné implementovat následující čtyři metody:

- rowCount() - vrací počet řádků
- columnCount() - vrací počet sloupců
- data() - vrací data
- headerData() - vrací data hlavičky tabulky (první sloupce a první řádek)

Delegate

Delegáta představuje třída *CellDelegate*, tato třída dědí z *QAbstractItemDelegate*, což je třída QT framworku určena pro vytvoření vlastního delegáta. Pro vykreslování položek bylo třeba implementovat tyto metody:

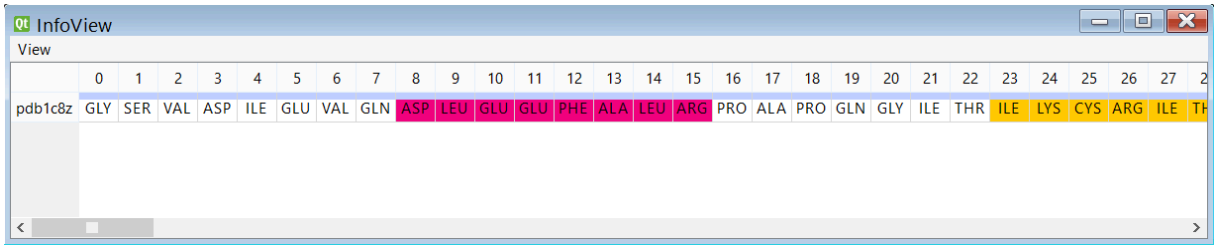
- paint() - definuje vykreslení položky - v tabulce je to v podstatě vykreslení jedné buňky
- sizeHint() - vrací rozměr vykreslované položky - v tabulce to je velikost jedné buňky, tato metoda se volá pro každou buňku v tabulce v případě, že se volá jedna z metod tabulky *resizeColumnsToContents* nebo *resizeRowsToContents* pro úpravu velikosti podle obsahu, při velkém počtu buněk tabulky je úprava velikosti buňky na základě obsahu pomalé a proto jsou použity pevně nastavené rozměry buněk tabulky

Pro každý režim zobrazovaných hodnot je definovány jiný obsah metody *paint*. Režimy byly implementovány tři:

- režim pro výpis sekvence aminokyselin s barevným rozlišením jednotlivých řetězců molekuly proteinu a barevným rozlišením pro prvky sekundární struktury jako je helix a strand, tento režim také umožňuje zvýraznit vybrané aminokyseliny ve 3D vizualizačním okně (*Viewer*)
- režim je výpis názvů jednotlivých atomů a jejich pozice jak byly definovány v PDB souboru
- režim slouží načtení vlastních dat a zobrazení v tabulce, pro tento režim je nutné mít vlastní data uložená v textovém souboru, kde hodnoty jedné buňky jsou na jednom řádků odděleny

mezerou a skládají se z těchto hodnot: řádek, sloupec, barva, text, souřadnice; barva se skládá ze tří hodnot 0-255; souřadnice jsou složeny také ze tří hodnot

Ukázka informačního okna (obrázek 4-3) s výpise aminokyselin, barevně jsou odlišeny ty, které tvoří sekundární strukturu.



4-3: InfoView okno

4.3 Ovládání scény

Ovládání 3D scény je základ vizualizačních aplikací. Pro zjednodušení implementace byla použita knihovna libQGLViewer, ve které jsou již všechny potřebné funkce implementovány. Mezi hlavní funkce, které tato knihovna řeší je ovládaní scény pomocí myší v kombinaci s klávesnici, jde o základní manipulaci se scénu-rotaci, translaci, zoom, ale také např.: přiblížení vybrané oblasti. Dále knihovna řeší výpočet optimální vzdálenosti ořezových rovin scény. Knihovna je založena na Qt Frameworku, při vytváření třídy vizualizačního okna *Viewer* stačilo použít třídu *QGLViewer* této knihovny jako bázovou. Jak už bylo uvedené, knihovna řeší ovládání scény. Jeden z cílů aplikace byla implementace funkce, která by umožňovala transformaci scény (rotaci, translaci) jednoho okna aplikovat i na jiné okno. Výsledkem by tak mělo být, že uživatel zkoumající dvě molekuly nemusí rotovat každou molekulou zvlášť, ale může rotovat oběma najednou. Implementace této funkce bude popsána v následující podkapitole 4.3.1 nazvané společná kamera.

4.3.1 Společná kamera

V této části bude popsaný postup jak aplikovat transformaci (posun, rotace scény) použitou na jednom okně do okna druhého. První bude uveden popis jak se vlastně transformace aplikují a dále bude uveden postup pro sdílení transformací.

Hlavní třída je *QGLViewer*, což je widget OpenGL okna - zajišťuje vykreslení scény a zpracovává události vyvolané klávesnicí nebo myší. Součásti *QGLViweru* je instance třídy *qglviewer::Camera*, která sestavuje před vykreslením projekční a modelview matici OpenGL. Modelview matici sestavuje na základě pozice a rotace kamery (pozorovatel scény). Základní třída reprezentující souřadnicový systém definovaný pozici a rotaci je *qglviewer::Frame*, od této je odvozena další třída *qglviewer::ManipulateFrame*, která může být ovládána myší - posun souřadného systému nebo rotace souřadného systému. Dále je definována ještě třída *qglviewer::ManipulateCameraFrame*, se kterou pracuje třída *qglviewer::Camera*. Při posunu scény (posun myši se stisknutým pravým tlačítkem nebo scrollování) se vytvoří událost, kterou *QGLViewer* zasílá do *Frame* objektu *Camera*, kde se zpracovává. V případě, že šlo o posun tak vypočte vektor posunu o který se kamera posune (změna pozice v objektu *Frame* kamery). V případě, že se jedná událost vyvolávající rotaci (posun se stisknutým levým tlačítkem myši) vypočte se úhel a vektor představující rotaci (kvaternion), který se nastaví objektu *Frame* kamery a poté se následně rotace provede voláním metody spin().

Pro sdílení transformací bylo třeba uchovávat instance vytvořených Frame spojených s kamerami a v případě, že se v některém zavolá metoda *translate* nebo *spin* tak tyto transformace aplikovat i na ostatní *Frame*.

Prvním krokem bylo vytvoření třídy *MyFrame*, která vychází z třídy *qglviewer::ManipulateCameraFrame*

```
class MyFrame:public qglviewer::ManipulatedCameraFrame
{
public:
	MyFrame();
	void translate(qglviewer::Vec& t);
	void spin();
	static void ResendEvents(bool b) {resendEvents = b;}
	static QList<MyFrame *> allFrames;
	static bool resendEvents;
};
```

Ve třídě bylo nutné zastínit metody *translate* a *spin* - byla potřeba upravit definici metody *translate* v souboru frame.h přidáním klíčového slova virtual, čímž je zajištěno, že se budou volat metody definované ve třídě *MyFrame*. Metoda *spin* definována v ManipulatedCameraFrame.h již jako virtual definována byla.

Funkce třídy je následující. Při vytváření objektu je jeho instance přidána do pole *allFrames* - toto pole slouží k rozeslání transformace ostatním instancím (jiné okno s molekulou), dále je zde definována proměnná *resendEvents* typu bool, která určuje zda se ostatním instancím budou události rozesílat. Změna proměnné *resendEvents* je řešena v události *keyPressEvent(QKeyEvent *e)* třídy *Viewer* vyvolané při stisku klávesy "x". Dále byly implementovány popisované metody *translate* a *spin*.

Do metody *translate* se posílá jako parametr vektor, o který se má změnit pozice - tento vektor však odpovídá souřadnému systému aktuálního frame, aby se mohla volat metoda *translate* i pro ostatní frame bylo potřeba vektor posunu nejprve přetransformovat do světových souřadnic (world coordinate) a poté na vektor aplikovat transformaci rotace každého frame a takto upravený vektor každému poslat voláním rodičovské metody *translate*.

Metoda *rotate* funguje podobně, nepřeposílá se vektor, ale kvaternion. Stačilo tedy zjistit kvaternion aktuálního frame - metodou *spinningQuaternion()*, tento kvaternion nastavit ostatním frame metodou *setSpinningQuaternion()* a poté zavolat metodu *spin()* rodičovské třídy, která rotaci provede.

Při vytváření instance vizualizačního okna a tedy instance třídy *Viewer*, která je jeho součástí bylo nutné pro zajištění funkčnosti aplikování transformací mezi okny v konstruktoru třídy *Viewer* nahradit objekt *Frame* kamery, popisovaným objektem *MyFrame*.

4.4 Vykreslování

V této části budou uvedeny podrobněji kapitoly související s vykreslováním. V první části je představeno OpenGL, pomocí kterého je řešeno vykreslování. Dále je popsána hlavní vykreslovací metoda implementované aplikace. Následující část popisuje použité shadery a jejich implementaci, hlavně jde o vykreslení koule a válce jako sprite. Další části jsou o sestavování vertexových polí a různých reprezentací molekul. U reprezentace cartoon je detailněji popsané generování trojúhelníkové sítě, pomocí které je tato reprezentace molekuly vykreslena.

4.4.1 OpenGL

Jedná se o API, pomocí kterého lze využít grafickou kartu k vykreslování 3D objektů. OpenGL je hodně rozsáhlé a umožňuje využívat mnoho funkcí k dosažení efektivního vykreslování nebo k dosažení co nejlepšího výsledku po grafické stránce. V této části budou uvedeny tedy jen funkce, které byli použity při implementaci aplikace k vizualizaci molekul proteinů.

Vertex Arrays (VA)
Vertexová pole slouží k popisu vykreslovaných objektu scény. Každý objekt ve scéně je popsaný vertexy a ty tvoří polygony. Poskládáním polygonu vznikají 3D objekty. Vertexová pole jsou tedy pole pro popis jednotlivých vertexů a jejich parametrů (např.: souřadnice, barva, normála). Hlavní výhoda tohoto popisu scény je, že po vytvoření těchto polí se ukazatele na tyto pole předají OpenGL a poté se může objekt vykreslit voláním jediné funkce.

Vertex Buffer Object
Tato technika umožňuje ukládat vertexová pole přímo do paměti grafické karty.

Blending
Pomocí techniky blending lze kombinovat barvu dvou pixelů na základě jejich hodnot alfa kanálu. Tuto technika lze využít např.: k vykreslování průhledných objektů nebo k odstranění aliasingu.

Shadery
Shadery jsou malé programy napsané v GLSL (OpenGL Shading Language), pomocí kterých lze definovat vlastní zpracování vertexů a fragmentů. Pomocí těchto programů lze vytvářet různé efekty, které by se pomocí fixní pipeline OpenGL nedaly provést.

[15]

4.4.2 Hlavní vykreslovací metoda

Při vykreslování se vycházelo z toho, že aplikace by měla umět zvýraznit různé části molekuly proteinu podle výběru uživatele, tedy vykreslovaný objekt by se měl umět vykreslit celý a nebo po částech. Za tímto účelem byla vytvořena bázová třída *GraphicObject*, ve které byli definovány dva typy virtuálních metod - k vykreslení celého objektu a k vykreslení části objektu (části odpovídají residuím řetězců - do vykreslovací metody se zasílá pole indexů residuí nebo pole intervalů residují). Pro různé grafické reprezentace jsou poté vytvořeny třídy, které dědí ze zmiňované třídy *GraphicObject* a přetěžují vykreslovací metody.

V hlavní vykreslovací třídě *Viewer* je poté definováno pole s objekty třídy *GraphicObject*, které se vykreslují. Pole objektů je zvoleno z toho důvodů, že některé grafické reprezentace se vykreslují ze dvou částí (např.: u reprezentace balls and sticks je výsledná reprezentace složena z koulí představující atomy a válců představující vazby), pro které jsou použity jiné techniky vykreslování (jiné shadery, jinak složená vertexová pole).

Hlavní metoda vykreslování je metoda *draw()* třídy *Viewer*. V této metodě jsou implementovány dva způsoby vykreslování, který z nich se použije záleží na tom zda uživatel zobrazuje celý model molekuly nebo zobrazuje molekulu se zvýrazněnými částmi. První způsob vykreslování volá metodu *draw()* pro všechny objekty určené k vykreslení, tedy se vykreslí vždycky celá část molekuly. Druhý způsob, kdy je potřeba zvýraznit vybrané části molekuly se vykreslují nejprve vybrané části - všem objektům k vykreslení se v metodě *draw* předává pole indexů residuí jímž odpovídající vertexy se mají vykreslit. Vykreslený výsledek se z color bufferu uloží do textury, v depth bufferu zůstane otisk vykreslených vybraných části, color buffer se vymaže a nyní se pro všechny objekty k vykreslení volám metodu *draw* s parametrem pole úseků označující nevybraná residua, do color bufferu se vykreslí jen části molekuly, které jsou před původně vykreslenýma vybranými částmi - blíže pozorovateli, obsah color bufferu se uloží do druhé textury. Nyní se vymaže color buffer a pomocí shaderu se vykreslí obdélník přes plochu celého okna. Úkolem fragment shaderu je zkombinovat tyto dvě textury, tak aby nevybrané části molekuly skoro splývaly s pozadím a vybrané části byly výrazně viditelné. Barva fragmentu je tedy kombinací 3 vrstev (barva pozadí, textura s vybranými částmi, textura s nevybraným částmi).

Popis kombinace

Proměnné barev (RGBA), které se budou kombinovat jsou:

- cs - barva z textury s vybranými částmi molekuly
- cu - barva z textur s nevybranými částmi molekuly
- cb - barva pozadí
- col - výsledná barva - výsledek kombinace

Kombinace probíhá na základě alfa kanálu barev *cs* a *cu.*

Jestliže nebylo nic vykresleno do textury s vybranými částmi molekuly (alfa kanál je nulový- cs.a=0), tak se bude kombinovat jen barva pozadí (cb) s barvou z textury s nevybranými částmi (cu). Výsledná barva (col) se vypočte:

$$\boldsymbol{col = \left(cb_{rgb} * (1 - cu_a) + cu_{rgb} * cu_a\right) * 0.3 + cb_{rgb} * 0.7}$$

Jestliže nebylo nic vykresleno do textury s nevybranými částmi molekuly (alfa kanál je nulový- cu.a=0), tak se bude kombinovat jen barva pozadí (cb) s barvou z textury s vybranými částmi (cs). Výsledná barva (col) se vypočte:

$$\boldsymbol{col = cb_{rgb} * (1 - cs_a) + cs_{rgb} * cs_a}$$

Jestliže se vykreslilo do obou textur, bude se kombinovat barva pozadí (cb), barva z textury s vybranými částmi (cs) a barva z textury s vybranými částmi (cu). Výsledná barva (col) se vypočte:

$$\boldsymbol{col = \left(cb_{rgb} * (1 - cs_a) + cs_{rgb} * cs_a\right) * 0.7 + (cb_{rgb} * (1 - cu_a) + cu_{rgb} * cu_a) * 0.3}$$

Konstanty 0.7 a 0.3 určují poměr intenzit barev ve kterém budou ve výsledku zahrnuty vybrané části molekuly a nevybrané části molekuly, případně barva pozadí a barva nevybrané části molekuly.

4.4.3 Shadery

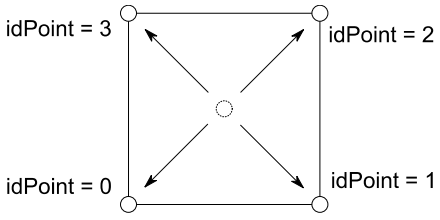
Při vykreslování zakulacených části molekul (koule, válec) nastal celkem výrazný problém při vykreslování. Vykreslení koule není v OpenGL jednoduchá operace. Při klasickém vykreslování koule nebo jiného 3D objektu, vycházíme z toho, že povrch objektu lze reprezentovat síti trojúhelníku. V případě hranatých 3D objektů to není zase tak velký problém jako u objektů se zakulaceným povrchem jako je koule, protože čím pěknějšího vzhledu koule chceme dosáhnout, tím více musíme použít trojúhelníku, které aproximují povrch koule. Pro scénu s velkým počtem koulí jako je molekula proteinu, kde každý atom je reprezentován jednou kouli je tato metoda neefektivní. Jinou alternativou vykreslování je vykreslovat koule jako sprite (billboarding nebo také impostor) - tato metoda vykresluje objekt jako jednoduchý obdélník s texturou, natočený vždy ke kameře. Tato technika však nelze použít s fixní pipeline OpenGL a proto bylo třeba vytvořit vlastní shadery, které budou popsány v následujících části. Zároveň byl vytvořen i další shader ke kombinaci textur, který je použitý ke zvýraznění části molekul v kapitole 4.4.2. Dále byly vytvořeny shadery, na základě příkladu shaderu aplikace RenderMonkey [30], s nimiž vypadají vykreslené model jako by byli z plastu, Zdrojové kódy shaderů byly uloženy ve zdrojích aplikace (Resource) - přímo v spustitelném souboru aplikace.

4.4.3.1 Vykreslení koule jako sprite

V této části bude popsána implementace shaderu vykreslující kouli na základě zdroje [6] (aplikace QuteMol). Implementace této techniky je rozdělena na dvě části. První část se týká vertex shaderu, jehož cílem bylo vytvořit čtverec natočený ke kameře. Ve druhé části, kterou řeší fragment shader bylo cílem vytvořený čtverce upravit tak, aby vykreslený výsledek vypadal jako koule.

Funkce vertex shaderu

Technika sprite vykresluje objekt jako čtverec natočený ke kameře, tedy pro jeden objekt (3D kouli) je potřeba použít čtyři vertexy. Pozice těchto vertexů se počítá ve vertex shaderu na základě pozice koule (atomu), která je pro čtyři vertexy zaslána do vertex shaderu stejná. Zároveň bylo třeba jednotlivé vertexy rozlišit a upravit jejich pozice, aby tvořili čtverec natočený ke kameře. Rozlišení jednotlivých vertexů je zajištěno parametrem *idPoint*, zasílaným do vertex shaderu s každým vertexem. Hodnoty parametru se pro čtyři vertexy tvořící jednou koulí nastaví na hodnoty: 0, 1, 2, 3. Na základě tohoto parametru víme, který roh čtverce bude vertex představovat a můžeme ho na danou pozici posunout přičtením vektoru, který vertex posune do rohu (obrázek 4-4).

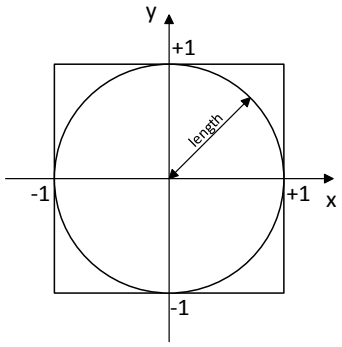


Obrázek 4-4: Přesun vertexů do rohu na základě parametru *idPoint*

Úkolem vertex sahderu je také nastavení hodnot, které se budou posílat do fragment shaderu (barva - *vColor* a poloměr koule - *vR*). U poloměru koule je důležité jej vynásobit konstantou *scale*, násobení zajišťuje, že atomy nebudou mít stejné poloměry např.: při přiblížení nebo oddálení. Hodnotu *scale* se posílá jako uniformní proměnná do vertex shaderu a je rovná velikosti bázového vektoru určující směr x-ové osy z aktuální modelview matice OpenGL. Dále se provádí transformace pozice vertexu do souřadného systému kamery a k transformovanému bodu se přičte vektor (posun do rohu) a nastaví se proměnná *vFragPos* na základě hodnoty proměnné *idPoint*, takto vzniklý bod (roh čtverce) se násobí s projekční matici.

Po vytvoření polygonu (čtverce) z vertexů následuje rasterizace, tedy plocha odpovídající polygonu se rozdělí na fragmenty, kde jeden fragment odpovídá 1 pixelu na monitoru. Jednotlivé fragmenty se dále posílají do fragment shaderu, ve kterém se může upravit barva fragmentu (pixelu) a hloubka fragmentu nebo jej můžeme odstranit.

U proměnných, které se posílají do fragment shaderu dochází k interpolaci mezi hodnotami definovanými v jednotlivých vertexech. To je využito k popisu pozice fragmentu ve fragment shaderu. Na základě pozice v polygonu (čtverci) je lze upravovat. K tomu je použita proměnná *vFragPos* což je 2D vektor, jehož hodnoty se nastaví na krajní hodnoty (-1,1). Interpolací se budou jeho hodnoty ve fragment shaderu pohybovat v intervalu <-1,+1>. Na obrázku 4-5 je vidět polygon a krajní hodnoty složek vektoru určujícího pozici fragmentu. Na obrázku je také vidět kruh, který chceme z polygonu ponechat.



Obrázek 4-5: Ořezáni polygonu

Popis fragment shaderu

Cílem fragment shaderu bude z jednotlivých fragmentů čtverce ponechat jen ty tvořící kruh (obrázek 4-5) a u nich vypočítat hodnotu hloubky (depth value) a barvu, tak aby vznikl dojem, že byla vykreslena 3D koule.

Nejprve se vypočte vzdálenost fragmentu od středu (*len*), v případě že vzdálenost je větší než jedna fragment se odstraní, tímto je zajištěno že z původního čtverce zůstanou fragmenty tvořící kruh. K výpočtu jsou použity složky vektoru *vFragPos*.

$$len = \sqrt{x^2 + y^2}$$

Dále se vypočte normála, souřadnice: x, y znám z pozice fragmentu, zbývá vypočítat souřadnici-z. Souřadnice-z normály (*zn*) je závislá na vzdálenosti fragmentu od středu, lze ji tedy vypočítat:

$$zn = \sqrt{1 - len^2}$$

Normála je použitá k výpočtu barvy fragmentu. Výpočet barvy vychází z Phongova modelu osvětlení [25], kde výsledná barva se skládá ze součtu ambientní, difuzní a zrcadlové složky. Jednotlivé složky jsou vypočteny následovně:

ambientní složka**:** $I_a = C * I_{amb}$
difuzní složka**:** $I_d = (n . l) * C * I_{dif}$
zrcadlová složka: $I_s = (rl . v)^h * C * I_{spec}$

C - hodnota barvy posílaná do shaderu, při výpočtech jsou použity složky RGB
n - vektor normály
l - vektor směru ke světlu
v - vektor směru k pozorovateli
rl - směr odrazu světla, výpočet je podle vztahu rl = 2 * (n . l) * n - l
l_{amb} - určuje intenzitu ambientní složky ze vstupní barvy
l_{dif} - určuje intenzitu difuzní složky ze vstupní barvy
l_{spec} - určuje intenzitu zrcadlové složky ze vstupní barvy
h - určuje ostrost zrcadlového odrazu

Dále je nutné vypočítat hodnotu hloubky fragmentu. Při vykreslování 3D objektu jako sprite vzniká problém vykreslení protnutých 3D objektů v případě, že do sebe zasahují. Když by se nevypočítala nová hodnota pro depth test, tak by se v případě dvou blízkých objektů vykreslil vždy jeden sprite objekt v popředí a druhy za ním, nikdy by nedošlo k vykreslení protnutí jako u 3D objektů popsaných trojúhelníkovou/čtvercovou síti.

Hloubku fragmentu je tedy nutné vypočítat následovně. Pro daný fragment se vypočte bod, který odpovídá bodu koule ve 3D prostoru, pro kouli se daný bod vypočte jako součet středu koule s normálu vynásobenou poloměrem koule. Dále je nutné s bodem provést výpočty, kterými získáme správnou hloubku fragmentu, která by odpovídala 3D objektu (sprite má totiž ve všech bodech stejnou hloubku), provádí se transformace projekční maticí a následně normalizace (x,y,z souřadnice bodu jsou vyděleny homogenní složkou vektoru), hodnoty bodu by měli ležet v rozsahu -1 až +1, zbývá tedy převést do intervalu 0 až 1 (zb=0.5*z+0.5), se kterým pracuje depth buffer. Hloubková hodnota fragmentu je závislá na z-souřadnici vertexu, proto stačí všechny výpočty provádět jen se z-souřadnicemi (pozice koule, normála, transformace). [22] [23] [24]

4.4.3.2 Vykreslení válce jako sprite

Vykreslení válce je rovněž obtížné, protože se jedná o zakulacený povrch v jednom směru. Jako řešení byla opět použita technika vykreslování jako obdélník natočený ke kameře (sprite). Trojrozměrného efektu je dosaženo stínováním ve fragment shaderu podobným způsobem jak to bylo v předchozí kapitole zabývající se vykreslováním koule jako sprite.

Popis shaderu vykreslující válec

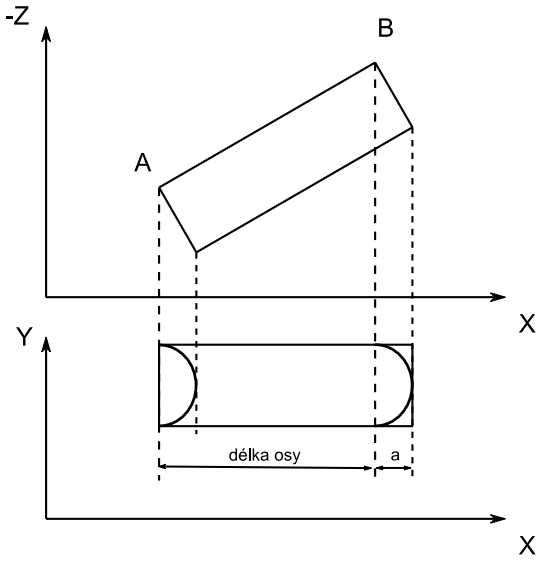
Válec se bude vykreslovat jako obdélník natočený ke kameře, pro každý válec se tedy budou do vertex shaderu zasílat čtyři vertexy s následujícími parametry:

- válec je dán osou určenou dvěma body, první bod (A) se posílá jako pozice vertexu a druhý bod je potřeba posílat jako parametr (B)
- poloměr válce (rad)

- identifikátor, na základě kterého se rozliší vertexy a umístí se do správných rohů obdélníku (idPoint)
- barva válce zasílána s vertexy

Ve vertex shaderu se body určující osu válce vynásobí s modelovací maticí, vzniknou tak dva body (AE, BE). Dále se počítá vektor určující směr osy válce jako rozdíl dvou bodů (dir=BE-AE). V případě, že z-složka směrového vektoru je kladná, body AE a BE se vzájemně prohodí a směrový vektor (dir) se vynásobí hodnotou: -1, tím se zajistí že všechny případy natočení válce mohu být zpracovány stejným postupem.

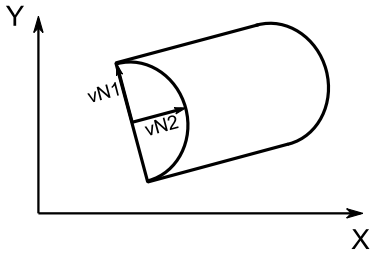
Dále se počítá šířka promítnutého válce na zobrazovací rovině. Velikost šířky je dána velikostí délky promítnuté osy válce (axisLen = length(dir.xy)) a délkou vzdálenosti (a = -dir.z*r; r je poloměr válce) znázorněné na obrázku 4-6:



Obrázek 4-6: Průmět válce

Poté se počítá osa válce ve zobrazované rovině XY, osa je dána složkami-X,Y směrového vektoru (osa = dir.xy), kolmici na osu získáme záměnou složek X, Y vektoru osy a změnou znaménka první z nich kolma = (-dir.y, dir.x).

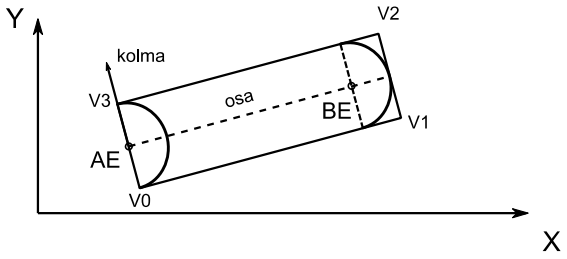
Dále se počítají vektory (vN1, vN2) potřebné pro výpočet normály ve fragment shaderu, jejich směry jsou zachyceny na následujícím obrázku:



Obrázek 4-7: Vektory, ze kterých počítám výslednou normálu pro osvětlení

vN2 = dir x (kolma.x, kolma.y, 0.0f)
vN1 = kolma

Pozici vertexů (V0-V3) se počítají na základě parametru *idPoint*, index vertexu odpovídá hodnotě *idPoint*.



Obrázek 4-8: Pozice vertexů

V0=AE-kolmaNorm*r
V3=AE+kolmaNorm*r
V1=BE-kolmaNorm*r+osaNorm*a
V2=BE+kolmaNorm*r+osaNorm*a

Vektory *kolmaNorm* a *osaNorm* jsou před výpočtem normalizovány. Konečnou pozici vertexu získáme vynásobením s projekční maticí.

Při výpočtu pozice vertexu se nastaví hodnota dvourozměrného vektoru *vFragPos*, jehož hodnoty se nastaví podle hodnoty *idPoint*, aby platilo že pro vertex:

V0: vFragPos=(0,-1)
V1: vFragPos=(1,-1)
V2: vFragPos=(1,1)
V3: vFragPos=(0,1)

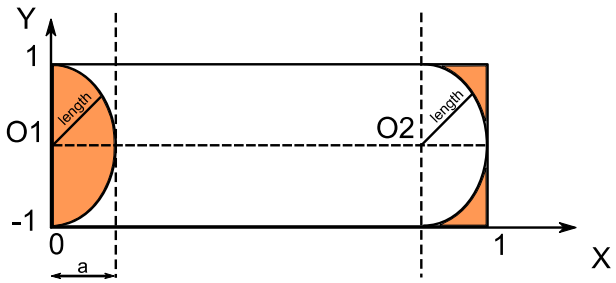
Vektor *vFragPos* je použitý k identifikace pozice fragmentu při zpracování ve fragment shaderu.

Popis fragment shaderu

Úkolem fragments shaderu je z obdélníku odstranit fragmenty, tak aby zbylé fragmenty tvořili průmět válce (přední poloviny válce). Na obrázku 4-9 jsou zvýrazněné části, které je potřeba odstranit. Na základě pozice fragmentu (*vFragPos*) se vypočte, ve které části polygonu fragment leží a zda se odstraní.

Nejdříve se porovnává x-složka vektoru *vFragPos* zda je větší než hodnota *pomLen* (představuje poměr délky promítnuté osy válce ku šířce celého obdélníku sprite). Na obrázku jde o odstranění rohů v pravé části obdélníku. Na oblast za touto hranici se lze dívat jako polokružnici, fragmenty ležící mimo ni jsou odstraněny. Od x-souřadnice pozice fragmentu se odečte hodnota *pomLen,* hodnota této souřadnice bude nyní v intervalu <0, *a*>, vydělením hodnotou a se dostaneme do intervalu <0,1>. Nyní lze použít souřadnici k výpočtu vzdálenosti fragmentu od pozice *O2*, na základě níž se fragment odstraním nebo dále zpracuje.

Poté se porovnává x-složku vektoru *vFragPos* zda je menší než hodnota *a=1-posLen*. Na obrázku jde o odstranění poloviny kruhu v levé části. Na oblast před touto hranici se opět dívat jako polokružnici, fragmenty ležící v kruhu jsou odstraněny.



Obrázek 4-9: Ořezání polygonu

Pro zbylé fragmenty se počítá normála, která je důležitá pro výpočet barvy a hloubky fragmentu. Normála je závislá na y-souřadnici vektoru *vFragPos* a na ose válce. Ve vertex shaderu byli vypočteny vektory *vN1* a *vN2,* které lze nyní použít k výpočtu normály. Vycházíme z jednotkové kružnice, pomocí které vypočteme příspěvek *vN2* do výsledné normály, příspěvek vN1 je dán velikosti y-složky vektoru *vFragPos*. Vztahy pro výpočet jsou:

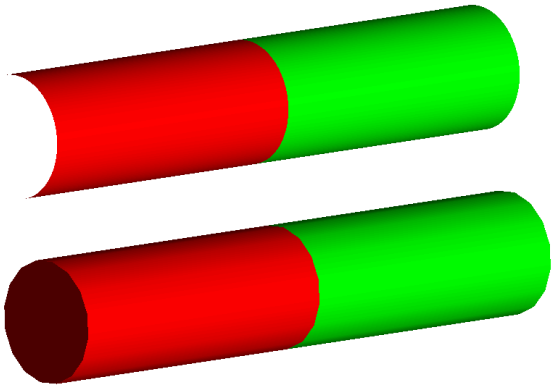
$$z = \sqrt{1 - y^2}$$

$$normal = z * vN2 + y * vN1$$

Výpočet barvy je stejný jako v případě koule, opět se skládá z ambientní, difuzní a zrcadlové složky, která se počítá na základě normály, směru ke světlu a směru k pozorovateli.

Výpočet hloubky fragmentu je podobný jako u koule. V tomto případě hledáme bod válce, který by se projekcí promítnul na pozici aktuálního fragmentu. Nejprve je potřeba nalézt bod na ose válce, který odpovídá aktuálnímu fragmentu a k němu přičíst normálu vynásobenou poloměrem válce. Dalšími výpočty (stejně jako u koule) získáme hloubku fragmentu.

Porovnání válce vykresleného jako sprite (obrázek 4-10 nahoře) a válce vykresleného jako mesh těleso (obrázek 4-10 dole). Na spodním válci jde vidět hrany hlavně na okrajích a také v polovině při přechodu mezi barvami. Mírně lze vidět na spodním obrázku podél celého válce pruhy - místa, kde jsou hrany. Výhoda vykreslení horního válce (sprite) je ta, že bylo použito pouze 8 vertexů.

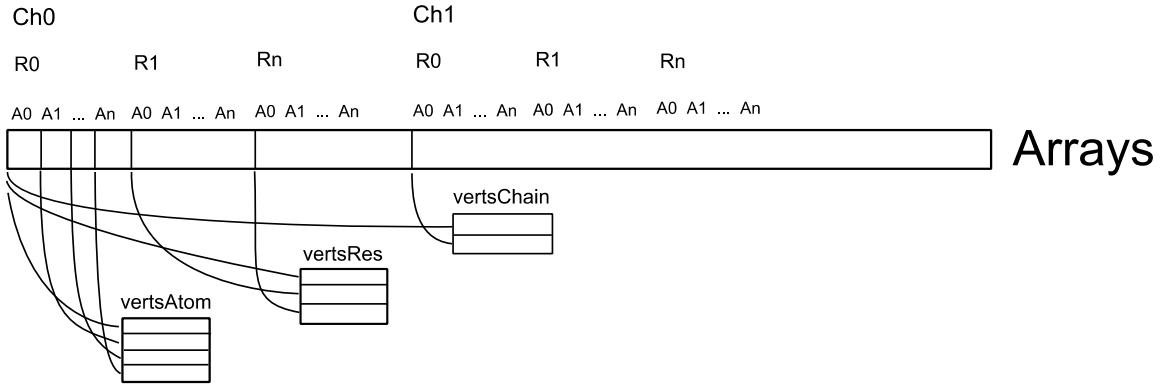


Obrázek 4-10: Porovnání vykreslení válců

4.4.4 Vykreslení různých reprezentací molekuly

Vykreslení je řešeno přes OpenGL. Snahou bylo dosáhnout co nejlepšího výkonu při vykreslování, proto je vykreslování řešeno pomocí VA (Vertex Array), efektivita této metoda spočívá v tom, že se pro každý vertex nemusí volat metoda pro nastavení barvy, normály a samotné pozice vertextu. Tyto hodnoty jsou uloženy v polích, ukazatele na tyto pole se pošlou do OpenGL a vykreslení celé scény se pak provede jedním příkazem. Další vylepšení spočívá v úspoře vertexu, protože není potřeba pro sousední polygony definovat v podstatě duplicitní vertexy s jejich parametry, ale stačí takové vertexy definovat jen jednou a polygony poté vykreslovat na základě indexů odpovídajícím daným vertexům - tato metoda se označuje Indexed Vertex Array. Další zrychlení přináší VBO(Vertex Buffer Objects), které umožňují ukládat data (vertex array) přímo do paměti grafické karty. Cílem tedy bylo vytvořit pole vertexů a ty poté uložit do VBO na grafickou kartu.

Metody vykreslování se liší podle vykreslovaného modelu (reprezentace) molekuly. V každém případě jde o sestavení VA. Při sestavování VA bylo nutné dodržet pořadí odpovídající prvkům datové struktury molekuly. Tedy VA začíná vertexy prvního atomu prvního residua prvního řetězce. V tako připravených polích lze jednoduše měnit barvu částí molekuly a tím odlišit různé části od sebe nebo je také zvýraznit na základě výběru uživatele. Pro využití takto vytvořených polí je potřeba si pamatovat indexy začátků jednotlivých části molekuly, to je řešeno pomocí polí pro jednotlivé struktury (atomy, residua, řetězce), do kterých se ukládají indexy vertexů z VA, kterými jednotlivé části molekuly začínají. Struktura pořadí je zachycena na obrázku 4-11. Také je na obrázku náčrt jak se ukládají indexy vertexů jednotlivých části molekuly do příslušných polí.



Obrázek 4-11: Sestavení polí

4.4.5 Reprezentace založena na vykreslování čar

Do této skupiny, kdy se molekula vykresluje pomocí čar spadá reprezentace wireframe. V této reprezentaci se vykreslují vazby mezi atomy čárou, kde jedná polovina odpovídá barvou jednomu atomu a druhá polovina je barvy druhého atomu. Jednotlivé vazby atomů bylo nutné rozdělit na poloviny a každou z nich vykreslit barvou odpovídající atomu, ze kterého vychází. Při sestavování VA je nutné dodržet pořadí, jaké bylo uvedeno v předchozí kapitole.

Pro vykreslení wireframe reprezentace se budou vertex pole skládat z pole vertexů (pozic) a pole barev. Pole vertexů bude obsahovat souřadnice krajních bodů polovin vazeb (každá polovina barevně odpovídá jinému atomu). Záznam v poli barev bude odpovídat barvě atomu, ke kterému vazba patří.

Pro sestavení VA bylo potřeba nejdříve projít všechny vazby a oběma atomům podílející se na této vazbě přiřadit vertex určující pozici uprostřed vazby. Postup byl následující:

Nejprve se struktuře atomu přiřadí vertexy uprostřed vazby s jiným atomem. K tonu byl použit kontejner standardní knihovny *std::map< Atom*, vector<Vec3D> > atomsVerts*, který umožňuje ukládat prvky jako dvojici klíč - hodnota. Klíč je ukazatel na strukturu atom a hodnota je dynamické pole *vector<Vec3D>* pro ukládání 3D souřadnic (Vec3D) - pozice uprostřed vazby. Na začátku je tato struktura prázdna a je potřeba ji nejprve naplnit před použitím. Naplnění probíhá tak, že se prochází všechny vazby molekuly a pro každou vazbu se vypočte pozice uprostřed:

$$X = (A1 + A2) * 0.5$$

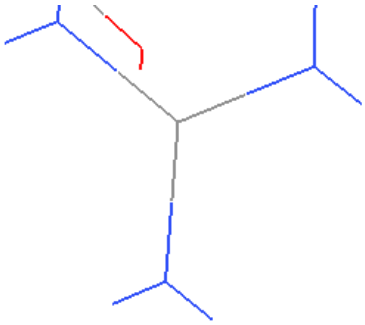
X je pozice bodu uprostřed vazby, A1, A2 jsou pozice atomů, mezi kterými je vazba.

Poté jsou atomy podílející se na této vazbě použity jako klíč do kontejneru *atomsVerts*, pomocí kterého se dá dostat k dynamickému poli každého atomu a do něj uložit pozici uprostřed vazby.

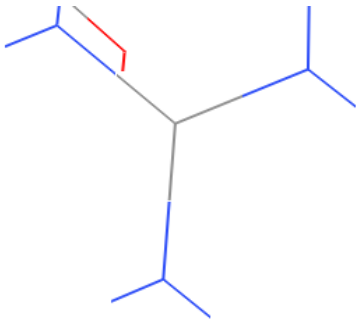
Po naplnění kontejneru *atomsVerts* lze sestavit VA. Postupně se prochází v cyklu jednotlivé řetězce molekuly, v dalším vnořeném cyklu se prochází residua daného řetězce a v dalším vnořeném cyklu se prochází jednotlivé atomy residua. Na úrovní atomů lze zjistit barvu tohoto atomu a použít ji do pole barev. Do pole s pozicemi vertexu se budou přidávat vždy dvojice vertexů. Dvojice se skládají z aktuální pozice atomu a pozice uprostřed vazby uložené v dynamickém poli v kontejneru *atomsVerts[atom]* (*atom* je ukazatel na strukturu aktuálního atomu). Počet dvojic je stejný jako počet hodnot v dynamickém poli pro daný atom. Pro každý záznam pozice se vytváří také záznam v poli barev s barvou odpovídající danému atomu. Počet záznamů ve VA polích musí být stejný.

Při vykreslování této reprezentace byla použita technika blending k vyhlazení čar (antialiasing). [16]

Ukázka vykreslení s použitím vyhlazení čar je na následujících obrázcích (4-12 a 4-13).



Obrázek 4-12: Bez antialiasingu



Obrázek 4-13: S antialiasingem

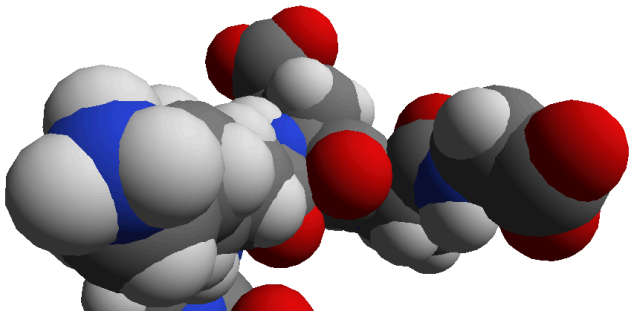
4.4.6 Reprezentace založené na vykreslování koulí a válců

V této části budou popsané všechny implementované reprezentace založené na vykreslování částí jako koulí a válců. K vykreslování byly použity shadery z kaptioly 4.4.3. Jednotlivé části jsou odděleny názvem popisované reprezentace.

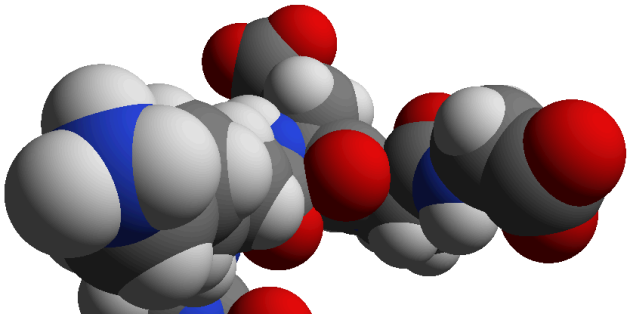
Spacefill

V režimu spacefill je každý atom molekuly reprezentován kouli s Van der Waalsovým poloměrem. Vykreslování v této reprezentaci bude probíhat s využitím shaderu pro vykreslování koulí jako sprite. Vertexová pole se budou skládat z pole vertexů, pole barev, pole s hodnotami float - poloměr koule a pole s hodnotami typu int - indexy rohů (parametr *idPoint*. Plnění polí probíhá tak, že se prochází struktura proteinu až na úroveň atomů a pro každý atom se do vertex polí přidávají vždy 4 položky, pro všechny 4 položky jsou hodnoty jednotlivých polí stejné až na výjimku pole s index rohů - ty obsahují pro jeden atom vždy hodnoty 0,1,2,3.

Porovnání klasického vykreslování atomů molekul jako koule (vykreslena pomocí metody *gluSphere(quad, 1.0f, 16, 16)* - koule aproximována trojúhelníkovou síti - obrázek 4-14) a vykreslování pomocí sprite obrázek 4-15 (4 vertexy na jednu kouli). Na obrázku 4-14 lze vidě hrany na okrajích koulí nebo také v místech průniků více koulí.



Obrázek 4-14



Obrázek 4-15

Balls and sticks

V tomto režimu jsou atomy vykresleny jako koule a vazby mezi atomy jako válce. Výsledný model je složený z dvou částí, kde v jedné se sestavují pole pro vykreslení atomů a v druhé se sestavují pole pro vykreslení vazeb. Další část se bude týkat sestavení polí pro vykreslení vazeb, vykreslení atomů je stejné jako v předchozí kapitole, jen byli zmenšeny poloměry koulí reprezentující atomy.

Při vykreslování vazeb jako válce byl použit shader, který vykresluje válec jako sprite. Sestavení polí pro vazby (válce) vychází ze vstupních parametru shaderu. Vertex pole se skládají z pole vertexů - 1.bod určující osu válce, pole barev, pole ve kterém se předává 2.bod určující osu válce a pole typu int, ve kterém se předává index rohu sprite-0,1,2,3. Vazby se vykreslují po polovinách, protože každá polovina odpovídá barevně atomu, ze kterého vychází. Pro každou polovinu se do každého pole přidávají 4 položky.

Sticks

V reprezentaci sticks se vykresluje model opět ze dvou části jako tomu bylo v reprezentaci balls and stick. Dalo by se říci, že je to v podstatě to samé jen pro koule představující atomy a válce představující vazby byly použity stejné poloměry. Ve výsledku vypadají vazby molekuly jako válce zakončené polokouli.

Backbone

V režimu backbone se vykresluje jen hlavní řetězec molekuly proteinu (main chain), konkrétně se vykreslují atomy N, CA, C, O pro každé residuum a jako válce se vykresluji vazby mezi těmito atomy. Jedná se tedy opět o zjednodušenou reprezentaci balls and sticks. Před sestavením polí bylo nutné vybrat zmiňované atomy hlavního řetězce a z nich poté pole pro vykreslování sestavit.

Trace

V této reprezentaci se vykresluji válce spojující atomy alfa uhlíků po sobě jdoucích residuí. Atomy alfa uhlíků se vykresluji jako koule se stejným poloměrem jako uváděné válce - tím je docíleno pěknějšího vzhledu napojení válců na sebe. Model výsledné molekuly se opět skládá ze dvou částí a postup sestavení polí je podobný jako u reprezentace sticks, pracuje se jen s atomy alfa uhlíků.

4.4.7 Reprezentace založené na vykreslování křivky

Tato část se bude týkat reprezentací tube a cartoon. Vykreslování obou reprezentací je založeno na stejném principu, kdy se podél křivky dané pozicemi alfa uhlíku vykresluje "potrubí". Pro reprezentaci cartoon se v úsecích řetězce se sekundární strukturou vykresluje místo "potrubí" pás, pro šroubovici (helix) se vykresluje pás zatočený do spirály, pro vlákno (strand) se vykresluje pás zakončený šipkou, natočení pásu by mělo být ve směru vodíkových vazeb.

Pro vykreslení molekuly proteinu v těchto reprezentaci bylo nutné sestavit trojúhelníkovou síť, popisující model proteinu v jedné z uvedených reprezentací. Vytvoření trojúhelníkové sítě se skládá z následujících části:

- **vytvoření křivky** - křivka bude sloužit jakou "osa", podél které bude trojúhelníková síť vytvořena
- **výpočet normál rovin** - v každém bodě křivky je nutné vypočítat normálu roviny, ve které budou ležet vertexy trojúhelníkové sítě

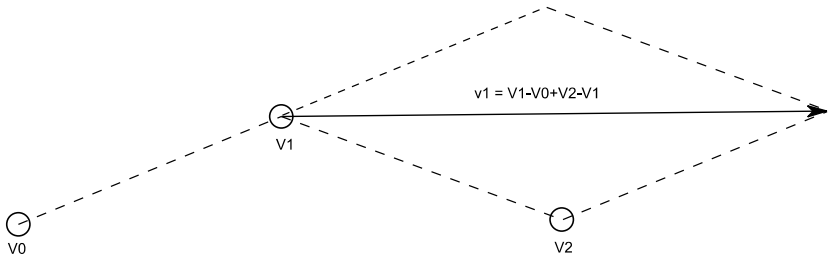
- **výpočet směru natočení pásu** - pro úseky řetězců, ve kterých se nachází prvky sekundární struktury (helix, strand) bude síť tvořena pásem a musí se tedy v těchto úsecích vypočítat směr natočení pásu
- **plnění vertexových polí** - v každém bodě křivky jsou vypočteny pozice vertexů a ty jsou přidány do vertexových polí, zároveň se provádí i triangulace mezi vertexy a vzniká tak požadovaná trojúhelníková síť

Pro reprezentaci tube je vynechána část s výpočtem směru natočení pásu, protože se v této reprezentaci pás nebude vykreslovat a výpočet by byl zbytečný.

Vytvoření křivky

K vytvoření křivky (osy) byla použita Hermitova kubika (Fergusonova kubika) [25], jedná se o interpolační křivku určenou dvěma řídicími body (počáteční a koncový bod křivky) a dvěma tečnými vektory v těchto bodech. Výhodou těchto kubik je jednoduché navazování - to je docíleno tím, že poslední bod segmentu je stejný jako počáteční bod následujícího segmentu. Spojitost C1 je docílena stejným tečným vektorem ve společném bodě.

Jako řídicí body křivky byly použity pozice atomů alfa uhlíků. Pro řídicí body bylo nutné vypočítat tečné vektory. K výpočtu tečných vektorů byli použity sousední řídicí body. Znázornění tečného vektoru v řídicím bodě je na následujícím obrázku 4-16.



Obrázek 4-16: výpočet tečného vektoru

Na obrázku 4-16 jsou řídicí body (V0, V1, V2 - pozice atomů alfa uhlíků po sobě jdoucích aminokyselin), tečný vektor je počítaný v bodě V1 jako součet vektoru směřující k bodu V1 z předchozího bodu (V0) a vektoru směřujícího k následujícímu bodu (V2), výsledný vztah pro výpočet je:

$$\overline{v1} = V1 - V0 + V2 - V1$$

Tento vztah se lze zjednodušit na tvar:

$$\overline{v1} = V2 - V0$$

Tečný vektor lze vypočítat jako rozdíl pozic bodu následujícího a bodu předchozího. Pro první a poslední bod je použitý nulový tečný vektor: $\bar{t} = (0, 0, 0)$.

Když jsou známy řídicí body a v nich definovány tečné vektory, mohu být vypočteny nové body ležící na interpolační křivce. Důležitým prvkem je počet generovaných bodů mezi uzly, vyšší počet

znamená jemnější trojúhelníkovou síť - výsledek bude pěknější, ale vykreslení bude náročnější na výkon. Souřadnice nového bodu je závislá na parametru t, vypočte se podle vztahu Hermitovské kubiky:

$$P(t) = V_0F_1(t) + V_1F_2(t) + \overline{v_0}F_3(t) + \overline{v_1}F_4(t)$$

V0, V1 jsou řídicí body a v_0, v_1 jsou tečné vektory definované v řídicích bodech

F1, F2, F3, F4 jsou Hermitovské polynomy:

$$F_1(t) = 2t^3 - 3t^2 + 1$$

$$F_2(t) = -2t^3 + 3t^2$$

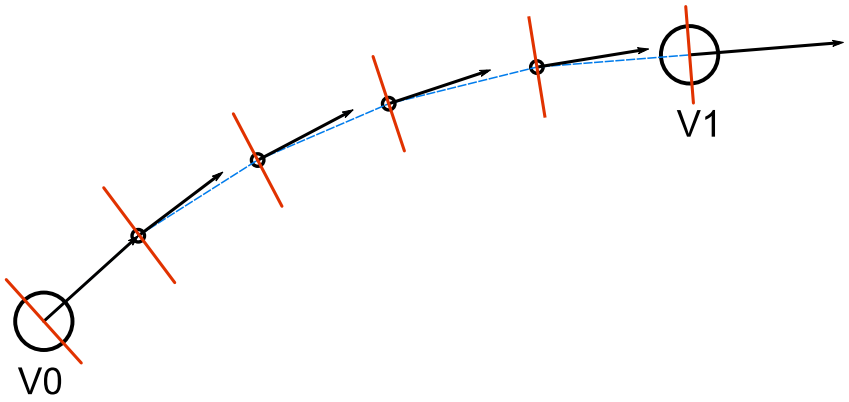
$$F_3(t) = t^3 - 2t^2 + t$$

$$F_4(t) = t^3 - t^2$$

[25]

Výpočet normál rovin

Na následujícím obrázku 4-17 je znázorněný úsek křivky mezi dvěma řídicími body V0 a V1. Pro každý bod je vypočtena normála (černá šipka) roviny (červená čára), ve které budou ležet vertexy pro jednotlivé části řetězce proteinu.



Obrázek 4-17: Znázornění rovin a jejich normál

Normály těchto rovin vypočtu z bodů křivky stejným postupem jako v případě výpočtu tečných vektorů interpolační křivky (str. 35), v tomto případě je potřeba vypočítat normály rovin i v krajních bodech. Pro první bod se vypočte jako rozdíl pozic druhého a prvního bodu, pro poslední bod se vypočte jako rozdíl pozic posledního a předposledního bodu.

Pro pás (helix, strand) budou v této rovině ležet vertexy obdélníků a pro zbytek řetězce budou v rovině vertexy n-úhelníku (aproximace kružnice). Vertexy po obvodu sousedních geometrických

útvarů se poté propojí obdélníky, tímto dojde k zaobalení křivky a vytvoření sítě pro vykreslení. Popis výpočtů pozic vertexů obdélníků a n-úhelníků bude podrobněji rozveden v dalších částech.

Výpočet směr natočení pásu

Pro vykreslení pásu - obdélníků, ze kterých je složený je potřeba pro každý bod křivky (osy) vypočítat směr natočení podle směru vodíkové vazby. Výpočet směru je podle zdroje [26], ve kterém byl použit směr od atomu alfa uhlíku k atomu kyslíku v hlavní části aminokyseliny, výhodou je že směr je vypočten z jednoho residua a není potřeba počítat se dvěma, mezi nimiž je vodíková vazba.

Vypočtený směr natočení je dále upraven, aby byl v rovině, ve které budu ležet vertexy trojúhelníkové sítě. Úprava spočívá v projekci vektoru směru na tuto rovinu a poté v normalizaci vektoru.

Nutné bylo kontrolovat směry natočení, mezi sousedními řídicími body, zda nesměřují oproti sobě opačnými směry [26]. Kontrola probíhá tak, že se počítá úhel mezi dvěma rovinami dané spojnici mezi řídicími body a jedním směrem natočení (1.rovina) a druhým směrem natočení (2.rovina). V případě, že je úhel mezi rovinami větší než 90°, tak je nutné druhý vektor určující směr natočení vynásobit -1, tím se získá vektor opačného směru (úhel bude potom menší než 90°).

Takto získané směry vodíkových vazeb (natočení pásu) jsou definovány jen pro řídicí body. Nutné bylo vypočítat směry natočení i pro body získané mezi řídicími body. Směry natočení pro tyto body se vypočítají interpolací ze směrů definovaných v řídicích bodech pro daný segment. Tedy pro dva řídicí body A, B a směry natočení D_A, D_B v těchto bodech, se směr natočení (D_X) pro dílčí body segmentu vypočte podle vzorce:

$$D_X(t) = D_A*(1-t) + D_B*t$$

parametr t určuje pozici dílčího bodu mezi řídicími body v intervalu <0,1>

Plnění vertex polí

Naplněné vertex polí probíhá tak, že se řetězec rozdělí na úseky podle typu sekundární struktury. Postupně se prochází jednotlivé úseky, kterým odpovídá i vytvořená křivka a podle typu sekundární struktury se generuje trojúhelníková síť. V úseku bez pravidelné sekundární struktury se sestavuje geometrie připomínající "potrubí", které se skládá z vyplněné kružnice, propojení kružnic a na konci zase z vyplněné kružnice. V části, kde se nachází helix se geometrie skládá z vyplněného obdélníku na začátku, poté z propojení sousedních obdélníků a na konci je vyplněný obdélník. Úsek, kde je strand se geometrie sestavuje podobně jako úsek pro helix, až na výjimku zakončení, které je ve tvaru šipky. Popis geometrie pro různé úseky je uveden v následujících částech.

Vyplněná kružnice

Při vytváření vyplněné kružnice se do VA nejprve přidá vertex uprostřed kružnice a poté vertexy po obvodu kružnice. Vertexy kružnice se nacházejí ve 3D prostoru, proto bylo nutné definovat parametry, na základě kterých se pozice vertexů vypočtou. Jedná se o střed kružnice a normálu roviny, ve které se budou vertexy kružnice nacházet. Tyto parametry byly vypočteny v předchozích částech (pro střed kružnice je použit bod křivky).

Další parametry jsou poloměr kružnice a počet bodů ležící po obvodu kružnici, které ji budou aproximovat. U kružnic je použitý ještě jeden parametr: vektor kolmý na normálu, na kterém bude ležet první vertex na obvodu, to je z důvodů, aby sousední kružnice měli vertexy v jedné rovině, tím zajistíme že při spojení sousedních kružnic nebude výsledný obal nijak zkroucený. Výpočet pozic vertexů je na základě zdroje [27].

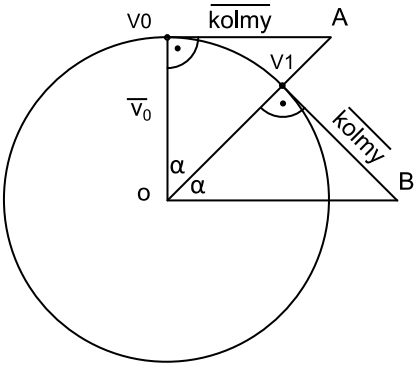
Vstupní parametry jsou:

- střed kružnice - O,
- normála roviny ve které leží kružnice - n
- vektor kolmý na normálu určující směr prvního vertexu - V0
- poloměr kružnice - r
- počet bodů po obvodu kružnice - *nVec*

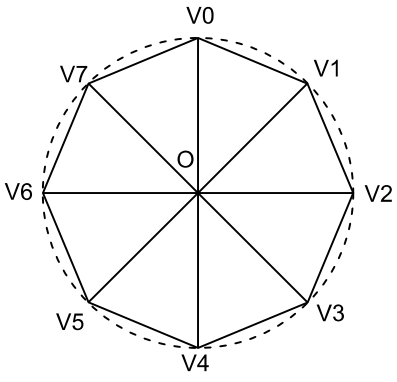
Výpočet pozic vertexů je znázorněn na obrázku 4-18, jedná se o zobrazení roviny ve které leží kružnice, normála této roviny směřuje ven z monitoru. První vertex je středový - je daný pozicí bodu křivky, další vertex se vypočte jako součet středového vertexu a vektoru v_0 určující směr prvního vertexu na obvodu kružnice, jeho velikost je normalizována a poté je násobený poloměrem kružnice.

$$V0 = O + v_0 * r$$

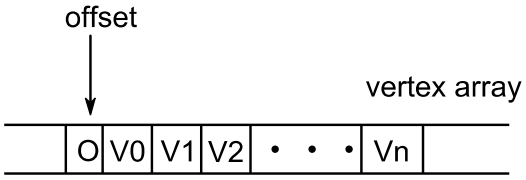
Další vertex na kružnici se vypočte následujícím postupem: vypočte se vektor kolmý na normálu roviny a vektor směřující od středu k předchozímu bodu $\overline{kolmy} = n \times (V0 - O)$, velikost kolmého vektoru se upraví na velikost rovnou: $\tan\alpha * r$. Součtem pozice předchozího bodu (V0) a kolmého vektoru se dostaneme na pozici bodu A, nyní upravíme velikost vektoru OA, aby odpovídala velikosti poloměru, vektor OA je vynásoben konstantou $\cos\alpha$ a tím dostaneme vektor OV1, přičtením k středovému bodu dostaneme pozici vertexu V1. Stejným způsobem vypočteme i pozice ostatních vertexů.



Obrázek 4-18: Výpočet vertexu



Obrázek 4-19: Triangulace



Obrázek 4-20: Plnění vertex polí

Přidávaní vertexů do vertex pole je zachyceno na 4-20, před výpočtem pozic vertexů kružnice se nastaví proměnná offset jako aktuální velikost vertex pole, poté se do pole přidá vertex středu kružnice (na pozici offset) a za ním následují vertexy po obvodu kružnice. Nyní je potřeba provést triangulaci (obrázek 4-19), z vytvořených vertexů se generuji trojúhelníky, které pokryjí plochu kružnice. Postupně se tedy přidávají indexy vertexů do indexového pole v následujícím pořadí pro 1. trojúhelník: O, V1, V2, indexy by se to zapsalo: offset, offset+1, offset+2. Postupně se tak přidávají i další trojúhelníky, vytvoření indexů provádí následující smyčka (nVec-počet vertexů na kružnici):

```
for(int i = 1; i<nVec; i++)
{
	addTriangle(offset, offset+i, offset+i+1);
}
addTriangle(offset, offset+nVec, offset+1);
```

S každým přidaným vertexem se také přidává normála (pro výpočet počítat osvětlení) do pole normál a také barva do pole barev. Při vyplněné kružnici jsou pro všechny normály stejné a jsou rovny opačnému vektoru určující rovinu, ve které leží kružnice.

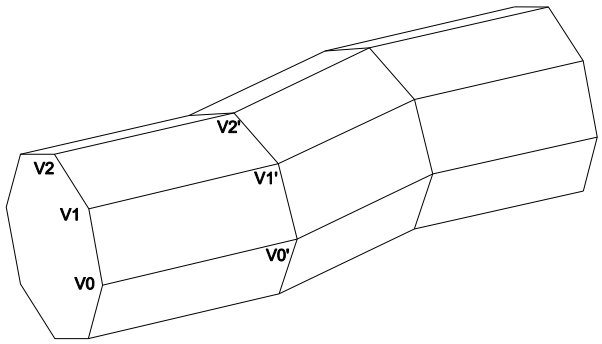
Vertexy kružnice

Do vertex polí se přidávají jen vertexy po obvodu kružnice, postup je podobný jako při vyplněné kružnici, ale nepřidává se vertex středu kružnice a nezapisují se indexy do indexového pole, také je rozdíl v normále použité pro osvětlení. Pro každý vertex je normála vypočtena jako rozdíl pozice vertexu a středu kružnice. S každým přidaným vertexem je takto vypočtena a přidána normála do pole normál.

Propojení sousedních kružnic - "potrubí"

Propojením vertexů sousedních kružnic se získá část geometrie, připomínající potrubí (obrázek 4-21). Vychází se z toho, že vertexy sousedních kružnic jsou uloženy ve vertex poli za sebou, jednotlivé obdélníky se vytváří přidáním indexů jejich vertexů do indexového pole (obdélníky jsou rozděleny na dva trojúhelníky a jejich indexy se přidávají do indexového pole). Podle obrázku 4-21 by se první obdélník sestavil z vertexů V0, V0', V1', V1. Podobně se vytvoří i další obdélníky z vertexů obou kružnic, kterými se zaobalí "křivka". Na obrázku 4-21 vypadá, že vykreslený výsledek bude hranatý, hrany ovšem nebudou vidět, protože v každém bodě je definována jedna

normála, výsledek se bude jevit zakulaceně jak po obvodu tak i podél křivky. Další možností ovlivnění výsledku je v množství propojovaných kružnic a také počtem bodů na kružnici.



Obrázek 4-21: Propojení sousedních "kružnic"

Vyplněný obdélník

Pro vygenerování vyplněného obdélníků je potřeba znát vstupní parametry pro obdélník, které jsou následující:

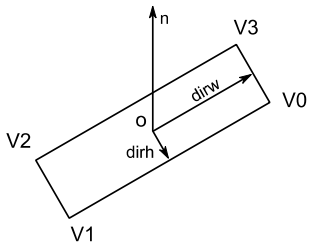
- normála určující rovinu, ve které bude obdélník ležet
- bod určující střed obdélníku
- vektor, určující směr jedné strany obdélníku - směr natočení

Výpočet jednotlivých vertexů je znázorněný na obrázku 4-22. Nejdříve je nutné vypočítat vektor určující směr druhé strany obdélníku (v tomto případě kratší strany-*dirh*).

$$dirh = dirw \times n$$

Vektory *dirw* a *dirh* jsou normalizovány a poté násobeny délkami jednotlivých stran. Pozice jednotlivých vertexů jsou poté vypočteny:

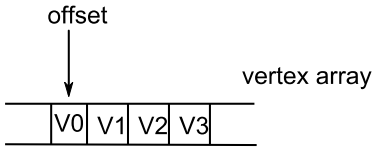
$$V0 = O + \frac{dirw}{2} + \frac{dirh}{2}$$
$$V1 = V0 - dirw$$
$$V2 = V1 - dirh$$
$$V3 = V0 - dirh$$



Obrázek 4-22: Vertexy obdélníku

Na obrázku 4-22 je vidět zadní strana obdélníku, tedy normála použita pro výpočet osvětlení je rovna opačnému vektoru znázorňující normálu roviny-n. Vektory *dirw* a *dirh* jsou zobrazeny v poloviční velikosti.

Vertexy jsou přidány do pole vertexů jak je zachyceno na obrázku 4-23:



Obrázek 4-23: Plnění vertex pole

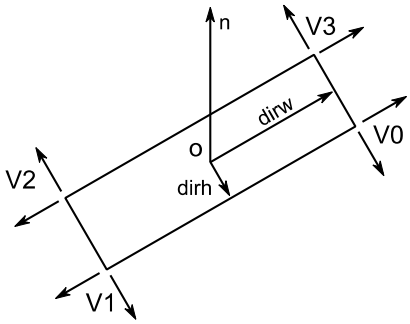
Dalším krokem je rozdělit obdélník na dva trojúhelníky (V0, V1, V2 a V2, V3, V0) a indexy odpovídající vertexům přidat do indexového pole. Trojúhelníky zapsané indexy by byli:

Offset, offset+1, offset+2	pro první trojúhelník
Offset+2, offset+3, offset	pro druhý trojúhelník

Vertexy obdélníku

Postup je podobný jako v případě vyplněného obdélníku, rozdíl je ve výpočtu normál pro osvětlení. Vertexy tvořící obdélník jsou umístěny v rozích, pro výpočet osvětlené je důležité v rozích definovat dvě normály, aby byla hrana při osvětlení viditelná, v každém rohu je tedy potřeba definovat dva vertexy s různými normálami (obrázek 4-24). Výpočet pozic vertexů je shodný, ale rozdíl je v přidání do pole vertexů, kdy se pozice vertexu přidá do pole vertexu dvakrát a do pole normál se přidávají různé normály. Pořadí přidávaní vertexů s normálami je následující:

1. vertexy V0, V1 s normálou = dirh
2. vertexy V1, V2 s normálou = -dirw
3. vertexy V2, V3 s normálou = -dirh
4. vertexy V3, V0 s normálou = dirw

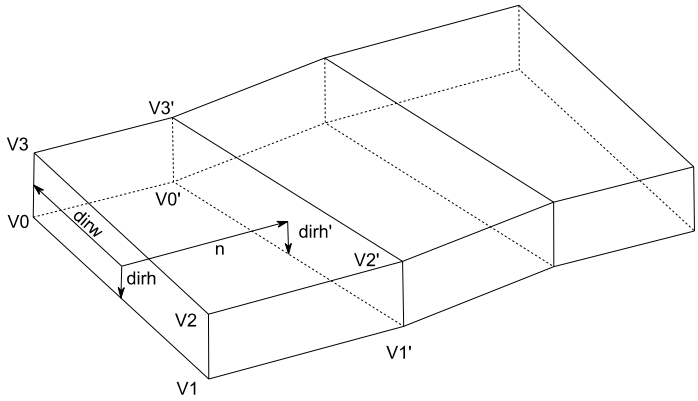


Obrázek 4-24: Směry normál pro výpočet osvětlení

Propojení sousedních obdélníků

Propojení obdélníků je podobné jako propojení sousedních kružnic. Ze dvou sousedních obdélníků (jejich vertexů) se vytváří obdélníky k vykreslení spojení těchto sousedních vertexů, jde o přidávání indexů vertexů do indexového pole. Část pásu vznikající propojením sousedních obdélníků je znázorněna na obrázku 4-25. V každém rohu obdélníků jsou definovány dva vertexy se shodnou pozicí, ale různou normálou, je tedy nutné spojovat správné vertexy. Pro vytvoření vrchního obdélníku je nutné spojit vertexy V2, V2', V3', V3, normála pro vertexy V2 a V3 musí být

-dirh. Pro vertexy V2' a V3' by normála odpovídala -dirh'. Vytvořené obdélníky jsou dále rozděleny na trojúhelníky, v obrázku již nejsou naznačeny pro přehlednost.

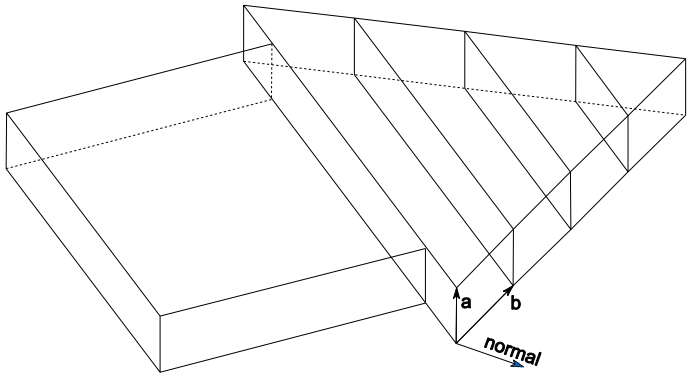


Obrázek 4-25: Propojení sousedních obdélníků

Šipka na konci vlákna (strandu)

Šipka je vykreslena podobně jako pás (propojení sousedních obdélníků), rozdíl je v tom, že při začátku vykreslování šipky je vytvořený obdélník s větší šířkou, než tomu bylo u obdélníků pásu. Dále se šířka dalších obdélníků zmenšuje. Důležitá změna je také při výpočtu normály pro osvětlení, tu je potřeba vždy znovu přepočítat. Nejprve se vypočtou vertexy dvou obdélníků sousedních a poté je se podle pozic vertexů počítají normály, výpočet je potřeba provést podél bočních vertexů šipky. Ukázka výpočtu je znázorněna na obrázku 4-26 u prvního vertexu prvního obdélníku, nejdříve se vypočtou vektory a,b z pozic vertexů a normála je výsledkem vektorového součinu těchto vektorů:

$normal = b \times a.$



Obrázek 4-26: Vytvoření šipky

4.5 Spuštění vytvořené aplikace

V této části jsou popsané postupy a požadavky ke spuštění vytvořené aplikace. Prvním způsobem je spuštění již zkompilované verze přiložené na CD. Druhou možností kompilace pomocí Visual Studia 2012, pro kompilaci je nutné mít nainstalovaný Qt Framework 5.1.0 s desktopovou verzí OpenGL a plugin Qt Frameworku do Visual Studia. Potřebné zdrojové kódy jsou přiloženy na CD.

Podmínkou pro spuštění a správnou funkčnost aplikace je 64-bitový operační systém Windows a grafická kartu, s podporou OpenGL verze 3.1.

5 Simulátor molekul

Tato kapitola bude popisovat jednoduchý simulátor molekul. Cílem bylo vytvořit jednoduchý simulátor molekul, který by pracoval na základě definovaného grafu představující atomy (uzly) propojené kovalentními vazbami (hrany) a pravidel chování atomů (působení sil v molekule), simuloval pohyb atomů. Postupně by se atomy měli z náhodného rozmístění uspořádat do 3D struktury, která by se podobala molekule proteinu. K uspořádaní by mělo dojít vyvážením působících sil mezi všemi atomy. Kapitola o simulátoru je členěna do sekcí, kde v první části je uveden rozsah vstupních dat - to je pro představu jak jsou molekuly proteinu rozměrné a s kolika atomy by tedy simulátor měl umět pracovat. V další části je vysvětlen principu fungování simulátoru. V poslední části je poté popis implementace simulátoru.

5.1 Rozsah vstupních dat

Vstupní data simulátoru molekul budou vycházet z reálných dat molekul proteinu. Ze statistiky databanky proteinu [14] obsahující necelých 92 tisíc proteinu je:

93% složeno z 23 atomů až 26 tisíc atomů
5% složeno z 26 tisíc atomů až 52 tisíc atomů
1.3% složeno z 52 tisíc až 77 tisíc atomů

Z 93% se nejčastěji vyskytují molekuly s počtem atomů do 15 tisíc. Ideální by bylo kdyby simulátor zvládal práci s tolika atomy (částicemi). Data ze statistiky jsou ze září 2013.

Dalším vstupem jsou kovalentní vazby, jejich množství bylo zjištěno načtením několika molekul proteinů, u nichž byly vazby vypočteny na základě vzdáleností dvojic atomů (kapitola 4.1). Výsledky jsou uvedeny v následující tabulce 5-1.

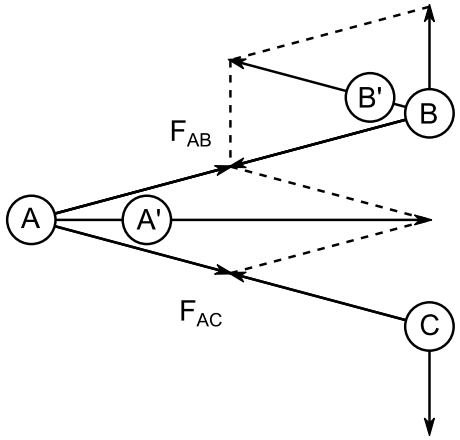
	Počet		
Protein	aminokyselin	atomů	kovalentních vazeb
pdb1a0s.ent	1239	9606	9849
pdb1aon.ent	8015	58674	59087
pdb1bpv.ent	104	1603	1624
pdb1c8z.ent	265	2213	2630
pdb1olh.ent	168	2792	2812
pdb1tba.ent	247	3881	3909
pdb1tii.ent	712	5469	5574
pdb2rb8.ent	93	723	736

Tabulka 5-1: Rozměry molekul proteinů

Počet kovalentních vazeb je přibližně stejný jako počet atomů. Náročnost simulace bude vycházet z počtu atomů.

5.2 Princip funkce

Simulátor bude fungovat v iteracích, kde v každé iteraci se pro každý atom vypočte síla, která na něj působí. Na základě velikosti této síly a směru působení se atom posune. Obrázek 5-1 znázorňuje působení sil mezi atomy. Atomy jsou vykresleny jako kolečka označené písmeny. Mezi dvojicemi atomů jsou šipkami znázorněny síly jak na sebe působí. Dvojice atomů AB a AC se přitahují - šipky směřují k sobě. Dvojice atomu BC se odpuzují - šipky směřují od sebe. Pro každý atom se tedy v jedné iteraci vypočte výsledná síla, která na něj působí jako součet všech dílčích sil z dvojic (na obrázku je výsledná síla znázorněna jen pro atomy A a B pro lepší přehlednost). Na konci iterace se atomy posunou o malý krok ve směru výsledné síly. Velikost kroku je také závislá na velikosti výsledné síly. Na obrázku jsou znázorněny posuny atomů A a B ve směru jejich výsledných sil, posun těchto atomů znázorňují atomy A' a B'. Na začátku další iterace budou nové pozice atomů výchozí a z nich se budou počítat nové působení sil mezi dvojicemi.



Obrázek 5-1: Znázornění sil a pohybu atomů

5.2.1 Pravidla chování atomů

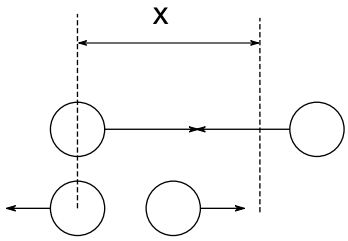
To jak se budou jednotlivé dvojice atomů ovlivňovat budou definovat pravidla. Pravidla by měli vycházet ze skutečných sil v molekule proteinu, to však vyžaduje hlubší znalosti z chemie. Zatím bylo definováno pár pravidel na základě sil stabilizující molekulu proteinu [2]. Cílem tohoto simulátoru mělo být, že by si pravidla nadefinoval sám uživatel - tato funkce zatím není implementována a je to jedna z možností další práce na tomto projektu.

Prvním pravidlem, které bylo definované je pravidlo pro atomy spojené kovalentní vazbou. Cílem tohoto pravidla bylo upravit vzdálenost atomů na vzdálenost rovnou délce kovalentní vazby. Bylo tedy nutné definovat funkci, která by na základě vzdálenosti dvou atomů vypočetla sílu jakou atomy na sebe působí. Směr působení je pro každou dvojici daný rozdílem jejich pozic. Funkce byla definována následovně:

$F = (length - 1.5) * covF$
length - aktuální vzdálenost mezi atomy
1.5 - požadovaná vzdálenost - délka vazby
covF - konstanta definující sílu kovalentní vazby

Funkce nejprve počítá rozdíl aktuální vzdálenosti atomů a požadované vzdálenosti. V případě, že aktuální vzdálenost je menší než požadovaná bude výsledek kladný - výsledná síla bude kladná a atomy se na konci iterace k sobě přiblíží. V opačném případě, že by aktuální vzdálenost byla menší než požadovaná výsledkem funkce by bylo záporné číslo - tedy i výsledná síla by byla záporná a

atomy by se odpuzovaly - na konci iterace by se od sebe vzdálili. Směr působení sil v závislosti na vzdálenosti znázorňuje obrázek 5-2. Když se atomy ustálí na požadované vzdálenosti (stejné jako je délka kovalentní vazby), tak výsledek funkce bude nulový a atomy se nebudou nijak ovlivňovat. Problém může být, když na takto ustálené atomy budou působit jiné atomy mnohem slabšími silami. Mohlo by dojít k rozhození již ustálených atomů (pořád jsou drženy silou kovalentní vazby). Proto byla pro každý atom definována nová proměnná - *staticForce*, která bude určovat jak těžké bude atom posunout. Na začátku iterace bude nastavena na hodnotu 1 a vždycky, když dojde k ustálení některé síly, tak se zvýší její hodnota. Na konci výpočtu sil bude výsledný síla působící na atom vydělena touto hodnotu - tím dojde k zabránění v pohybu (snížení kroku pohybu) již ustálených atomů.



Obrázek 5-2: Vzdálenost atomů a působení sil

Další pravidlo, by mělo fungovat podle Van de Waalsových sil. Cílem bylo ustálit vzdálenosti atomů na hodnotě 3.5 Å s vzdáleností větší než 4 Å se tyto síly hodně zmenšují proto se s nimi počítá jen u dvojic atomů vzdálených od sebe do 5 Å. Funkce byla použita podobná jako v prvním pravidle, jen s jinými konstantami:

$F = (length - 3.5) * vdwF$
vdwF - konstanta definující velikost síly

Dále by se hodilo definovat pravidlo, které by simulovalo vodíkové vazby - tyto vazby jsou velmi důležité u molekul proteinu, protože mají výrazný podíl na stabilizaci proteinového řetězce. Pravidlo zatím definováno nebylo, protože tato síla je elektrostatická a bylo by pravděpodobně nutné rozšířit parametry jednotlivých atomů o novou proměnnou. V této proměnné by byla uložena informace o náboji atomu, nutné by bylo tyto proměnné na začátku inicializovat a možná také v průběhu iterací aktualizovat - toto řešení by však vyžadovalo hlubší znalosti z chemie.

5.3 Aktuální stav programu

Pro každý atom jsou definovány tyto proměnné:
- pos (float3) - aktuální pozice atomu
- forces (float3) - součet všech sil působící na atom
- staticForce (float) - míra určující jak je atom držen na aktuální pozici

Na začátku programu je vygenerována náhodná pozice atomů, tak že jsou všechny u sebe a mohou na sebe všechny působit. Další varianta rozmístění, je rozmístit atomy podél osy-x se stejným krokem (0.2) a ve zbylých osách náhodně v intervalu <0;1>, čímž je zajištěno, že atomy jednotlivých residuí a sousední residua budou blízko sebe.

Simulátor pracuje v iteracích. V každé iteraci se procházejí všechny atomy, pro které se počítají síly jaké na ně působí, tyto síly se sčítají a na konci se z nich vypočte nová pozice atomu, vypočet pro jeden atom(A0) probíhá následovně:

-pro daný atom (A0) se vynuluje vektor *forces*

-pro daný atom se proměnná *staticForce* nastaví na hodnotu 1

-procházejí se všechny ostatní atomy - tak se projdou všechny atomy tvořící dvojici s A0, kde se pro každou dvojici vypočte směr působení síly (*dir*) - rozdílem pozic atomů s následnou normalizací, dále se testuje jestli jsou atomy spojeny kovalentní vazbou, na základě toho se vypočte síla (kapitole 5.2.1), jakou na sebe atomy působí. Normalizovaný vektor směru síly se vynásobí velikostí síly a vektor se přičte do proměnné *forces,* v případě, že jsou atomy v požadované pozici zvýší se hodnota proměnné *staticForce*:

$$forces += dir * F$$

-po projití všech dvojic s daným atomem (A0) by vektor *forces* měl být roven výsledné síle, nová pozice atomu se vypočte jako součet aktuální pozice a vektoru *forces*. Vektor forces je vydělen proměnnou *staticForce* (míra jak je atom držen na aktuální pozici) a poté je vynásobený parametrem *t* - ten určuje velikost kroku, s jakým se budou atomy pohybovat

$$pos = pos + (forces/staticForce) * t$$

proměnnou *t* se řídí velikost změny pozice (kroku)

5.3.1 Okno simulátoru

Simulátor byl přidaný do implementované vizualizační aplikace. Hlavní rozhraní, kterým se simulátor ovládá je okno *SimulView* (dědí z třídy *QMainWindow*). Menu okna *SimulView* umožňuje načtení PDB souboru, z kterého se načte struktura molekuly. Dále se skládá z vizualizačního okna *Viewer*, ve kterém lze pozorovat simulaci. Součástí jsou také GUI prvky jako pole pro změnu parametrů simulace (nastavení konstant *covF* a *vdwF* z kapitoly 5.2.1), tlačítka pro spuštění/zastavení simulace nebo pro resetování simulace - nastavení nových pozic atomů.

Vykreslování je řešeno pomocí časovače, který volá metodu překreslení *redraw()* každých 60ms. Toto řešení je z toho důvodů, aby se překreslování nevolalo každou iteraci a simulaci tak nebrzdilo. Další časovač je použitý k měření rychlosti simulace (počet iterací za jednotku času). Tento časovač se volá metodu *bench()* k měření výkonu každých 5 vteřin.

Součástí metody k měření výkonu je také výpočet podobnosti simulované molekuly a originálu načteného z PDB souboru. Porovnání spočívá ve vytvoření dvou polí vektorů pro simulovanou a originální molekulu z pozic atomů hlavního řetězce, porovnání těchto dvou polí se provádí kosinovou metrikou [29]. Pole vektorů pro molekuly se počítá z pozic atomů tvořící hlavní řetězec (N CA C). Vektory se počítají jako rozdíl sousedních atomů tohoto řetězce. První vektor by se vypočítal jako rozdíl pozic CA a N, druhý vektor by se počítal jako rozdíl pozic C a CA, třetí už by počítal s pozicí atomu následující aminokyseliny N od níž by se odečetla pozice atomu C.

5.3.2 Vlákno simulace

Samotná simulace (kapitola 9.4) po spuštění běží ve vlastním vlákně. Vlákno představuje třída *SimulWorker*, která dědí z třídy *QThread* (Qt framework). Ve třídě je implementována metoda *run(),* která se volá vždycky po spuštění vlákna. V této metodě se ve smyčce volají metody simulátoru v pořadí:

- *clearForce()* - pro všechny atomy vynuluje vektory, ve kterých je uložena síla působící na atom z předchozí iterace
- *computeForces()* - pro všechny atomy se počítají nové síly, které na ně působí
- *updatePos()* - vypočte se nová pozice atom

Součástí metody *updatePos(),* je také výpočet průměrné změny pozice atomu (*averageMove*). Na základě této hodnoty lze sledovat míru pohybů atomů a v případě, že je hodnota malá - síly v celé molekule se vyrovnaly a atomy se již skoro nepohybují, tak lze simulaci zastavit.

K urychlení běhu těchto metod je použito OpenMP API [28], které umožňuje části kódu zpracovávat paralelně na více jádrech procesoru.

6 Testování aplikace

Tato kapitola bude o měření výkonnosti aplikace. Nejdůležitější měření se bude týkat rychlosti vykreslování - tedy počtu vygenerovaných snímků za vteřinu (FPS - frame per second). Další měření se bude týkat paměťové náročnosti - budu měřit využití paměti (počítače i grafické karty). Měření se bude týkat všech režimu zobrazení, ve kterém se budou vykreslovat molekuly proteinu různých velikostí od nejmenší po největší (skoro 60 tisíc atomů). Dále se bude měřit rychlost vykreslován na základě různých rozlišení výsledného obrazu. Součásti této kapitoly budou také testy simulátoru, tedy kolik zvládne iterací za jednotku času.

Testování probíhalo na následující PC sestavě:

CPU	AMD Phenom II X4 840 3.2GHz
RAM	8GB RAM
GPU	AMD Radeon HD 6750
OS	Windows 8.1 64b

Tabulka 6-1: Testovací sestava

6.1 Rychlost vykreslování

V této části bude měřena rychlost vykreslování pro různá výstupní rozlišení obrázku. Měření je přidáno do samotné aplikace a probíhá tak, že se ve smyčce 500x za sebou volá metoda vykreslování a měří se čas vykonání této smyčky. Z naměřeného času se poté vypočtu průměrný čas potřebný k vykreslení jednoho snímku. Z průměrného času vykreslení jednoho snímku je vypočteno kolik snímku by aplikace zvládla vygenerovat během jedné sekundy (FPS). Rychlost vykreslování je měřena pro různé výstupní rozlišení, různé reprezentace a pro různě velké molekuly proteinů.

Závislost rychlosti vykreslování na rozlišení

V této části se měří rychlost vykreslení pro různá rozlišení výstupního obrazu. Měří se průměrný čas k vykreslení jednoho snímku a z něj je poté vypočtena hodnota FPS.

Rozlišení	Čas [ms]	FPS
800x600	0,61	1639
1024x768	0,672	1488
1280x960	0,802	1246
1600x1200	0,876	1141

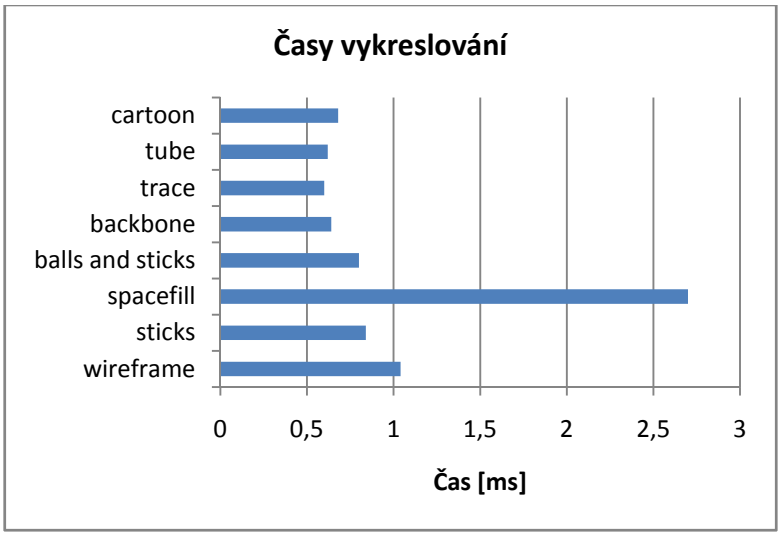
Tabulka 6-2: Rychlost vykreslování v závislosti na rozlišení

Závislost rychlosti vykreslování na reprezentaci

V této části se měří rychlost vykreslování molekuly proteinu 1A0S pro různé reprezentace, měření probíhalo pří výstupním rozlišení 1024x768.

Reprezentace	Čas [ms]	FPS
wireframe	1,04	961
sticks	0,84	1190
spacefill	2,7	370
balls and sticks	0,8	1250
backbone	0,64	1562
trace	0,6	1666
tube	0,62	1612
cartoon	0,68	1470

Tabulka 6-3: Čas vykreslování v závislosti na reprezentaci



Graf 6-1: Čas vykreslování v závislosti na reprezentaci

Závislost rychlosti vykreslování na velikosti molekuly proteinu

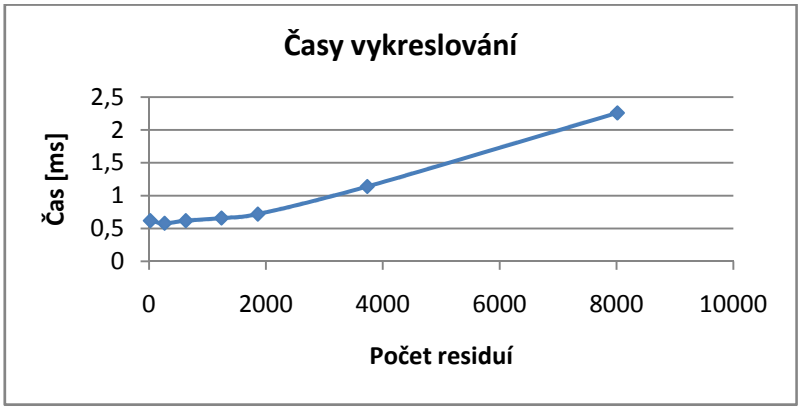
Pro tento test byly vybrány molekuly proteinů různých velikostí (tabulka 6-4), pro každou z nich se bud měřit průměrný čas vykreslení jednoho snímku, z čehož je poté vypočteno FPS - počet vytvořených snímků za sekundu. Měření probíhalo při rozlišení obrazu 1024x768 a vykreslovaná reprezentace byla cartoon. Scénu s molekulou byla upravena tak, aby molekula zabírala nejvíce prostoru ve vykreslovaném okně a aby byla celá vidět.

Počet	Molekuly proteinů						
	1l2y	**1c8z**	**4cd7**	**1a0s**	**4j0q**	**4nhz**	**1aon**
řetězců	1	1	2	3	5	16	21
reziduí	20	265	628	1239	1861	3734	8015
atomů	304	2213	5027	9606	14047	30090	58674
kovalentních vazeb	310	2491	5274	9849	14337	31237	59087

Tabulka 6-4: Parametry molekul proteinů

Protein	**1l2y**	**1c8z**	**4cd7**	**1a0s**	**4j0q**	**4nhz**	**1a0n**
Čas [ms]	0,62	0,58	0,62	0,66	0,72	1,14	2,26
FPS	1612	1724	1612	1515	1388	877	442

Tabulka 6-5: Čas vykreslování v závislosti na molekule proteinu



Graf 6-2: Čas vykreslení [ms] v závislosti na velikosti molekuly (počtu residuí)

Z výsledku měření rychlosti je patrné, že při zobrazení molekul proteinu v reprezentaci cartoon je výkon dostatečný, protože při největší testované molekule je výsledek 442 FPS. Optimální frekvence vykreslování by měla být 30-60Hz, aby se pohyb jevil plynulý. Z tabulky 6-3 se jeví jako nejnáročnější reprezentace na vykreslování spacefill, byl tedy proveden test i pro největší testovanou molekulu proteinu - 1a0n v reprezentaci spacefill. Výsledek měření byl 5,86 ms (170 FPS). K vykreslování by se dal použít i počítač se slabší grafickou kartou.

Aplikace omezuje rychlost vykreslování na 100 FPS. Přebytečný výkon se využije např.: při více otevřených oknech (porovnání molekul proteinů). Do budoucna by se také mohlo vylepšit vykreslování (dosažení pěknější výsledku, použití hustší trojúhelníkové síti), čímž by také mohlo dojít ke zvýšení nároků aplikace.

6.2 Paměťová náročnost

V této části bude popis měření spotřeby paměti vytvořené aplikace. K měření spotřeby paměti počítače byl použit správce úloh systému Windows. K měření využití paměti grafické kartě byla použita aplikace MSI Afterburner v2.3.1 [31]. Aplikace MSI Afterburner zobrazuje aktuální

využití paměti grafické karty, proto bylo nutné nejprve měřit využití paměti grafické karty před spuštěním implementované aplikace a tuto hodnotu poté odečíst od hodnot naměřených pro různé reprezentace/molekuly. Měření se bude týkat různých reprezentací a různě velkých molekul.

Využití paměti v závislosti na reprezentaci

V této části se měřilo využití paměti počítače i grafické karty pro různé reprezentace molekuly proteinu. K měření byla použita molekula proteinu 1a0s.

	RAM [MB]	GPU RAM [MB] aktuální hodnota využití	GPU RAM [MB] využití aplikací
Před spuštěním aplikace		135	
Spuštěná aplikace	7,6	139	4
wireframe	64,6	154	19
sticks	68,4	160	25
spacefill	65,2	162	27
balls and sticks	67,5	162	27
backbone	67,1	160	25
trace	65,1	160	25
tube	76,5	163	28
cartoon	70	162	27

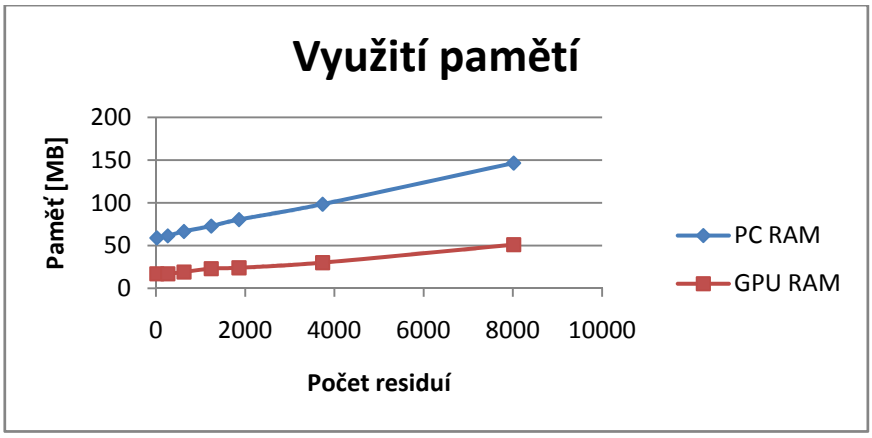
Tabulka 6-6: Využití paměti pro různé reprezentace molekul proteinů

Využití paměti v závislosti na velikosti molekuly proteinu

V této části je měření využití paměti počítače i grafické karty pro různě velké molekuly proteinů. Molekuly proteinů byly vykresleny v reprezentaci cartoon.

	RAM [MB]	GPU RAM [MB] aktuální hodnota využití	GPU RAM [MB] využití aplikací
Před spuštěním aplikace		145	
Spuštěná aplikace	7,6	149	4
1l2y	59	162	17
1c8z	61,4	162	17
4cd7	66,5	164	19
1a0s	72,8	168	23
4j0q	80,4	169	24
4nhz	98,4	175	30
1a0n	146,5	196	51

Tabulka 6-7: Využití paměti pro různé molekuly proteinů



Graf 6-3: Využití pamětí v závislosti na velikosti molekuly

Z měření se jako nejnáročnější na paměť ukázala reprezentace tube. Bylo provedeno tedy i měření spotřeby paměti v této reprezentace s molekulou proteinu 1a0n. Výsledná spotřeba GPU RAM byla 65 MB, spotřeba paměti počítače byla 172,5 MB. Využití paměti je přijatelné, když v dnešní době je skoro každý notebook vybavený 4 GB paměti a velikost pamětí grafických karet se pohybuje kolem 1 GB.

6.3 Testy simulátoru

Tato část popisuje měření rychlosti simulátoru. Měření se týká počtu vykonaných iterací během 5 vteřin. Rychlost simulace je závislá na počtu atomů a počtu pravidel, které se musí na jednotlivé dvojice aplikovat. V aktuální verzi jsou definovány dvě pravidla a testy byly provedeny na třech molekulách.

Počty vykonaných iterací na základě velikosti molekul je uvedeny v tabulce 6-8. Výsledky nejsou úplně přesné, protože hodnoty počtu iterací kolísaly.

	Molekuly proteinů		
	1l2y	**1c8z**	**1a0s**
Počet atomů	304	2213	9606
Počet iterací za 5sec	3160	85	4

Tabulka 6-8: Počet iterací v závislosti na velikosti molekuly

7 Závěr

Vizualizace

Cílem této práce bylo vytvořit aplikaci k vizualizaci molekul. Vizualizace měla být založena na OpenGL. Vytvoření aplikace se podařilo, výsledná aplikace zvládá vykreslení osmi grafický reprezentací molekul s využitím OpenGL. Jednou ze schopností aplikace mělo být vizuální porovnávání dvou molekul proteinů. Tedy výsledkem by mělo být, že uživatel zkoumající dvě molekuly nemusí otáčet nebo posouvat každou molekulou zvlášť, ale může použít transformaci na obě molekuly najednou. Dalším cílem bylo umožnění zvýraznění části molekuly proteinu, tedy aplikace by měl umožnit uživateli vybrat část proteinového řetězce a vybraná část by se projevila zvýrazněním vybrané části přímo ve vizualizaci. Tyto schopnosti byly při implementaci zohledněny. Vizualizace molekul proteinů probíhá v samostatných vizualizačních oknech. Také byla implementována funkce pro aplikaci rotace či posunu z jednoho okna na druhé. Základní funkce pro porovnávaní dvou molekul jsou zajištěny.

Při vykreslovaní některých reprezentací založených na vykreslování koule nebo válce vznikl problém, protože vykreslování těchto objektů je celkem náročná operace, zvláště když se v molekulách vykytují ve velkém počtu. Proto byla k vykreslování těchto objektů použita technika vykreslování - sprite. Tato technika si vyžadovala napsaní vlastních shaderu v jazyce GLSL, pomocí kterých se problém s náročnosti vykreslováním vyřešil a výsledek vykreslování byl mnohem pěknější.

Simulátor molekul

Jako rozšíření diplomové práce bylo cílem vytvořit jednoduchý simulátor molekul, který by umožňoval na základě definovaných pravidel simulovat pohyb jednotlivých atomů. Simulator se podařilo implementovat do funkční podoby i s vizualizaci. Zajímavé by bylo použít tento simulátor k simulaci skládaní molekuly proteinu. Zatím však nebyly popsány pravidly všechny důležité síly podílející se na stabilizaci proteinového řetězce. Simulované výsledky se tedy zatím molekule proteinu nepodobaly. Hlavní problém bude v definování pravidel chování atomů na základě sil působících v molekule, což vyžaduje hlubší znalosti z chemie.

Možnosti vylepšení aplikace

Možnosti rozšíření by mohly být například v podpoře dalších souborů s molekulami proteinu, či ve vizualizaci samotné. Dále by se dalo vylepšit GUI aplikace např.: použitím tlačítek pro přepínání mezi různými reprezentacemi vykreslování nebo pro různé barevné režimy. Další možnosti vylepšení by mohlo být nastavení aplikace, kde by se dali měnit různé konstanty používané v aplikaci, jako jsou například rozměry základních objektů používaných při vykreslování, pozice světla nebo poměr intenzit barev vybraných a nevybraných části molekuly. Úpravy v nastavení by se mohly ukládat do souboru a při spuštění aplikace načítat. Další vývoj aplikace by mohl pokračovat ve vylepšení simulátoru.

Literatura

[1] KODÍČEK, M. *bílkoviny*. From Biochemické pojmy : výkladový slovník [online]. Praha: VŠCHT Praha, 2007 [cit. 2013-12-27]. Available from www: <http://vydavatelstvi.vscht.cz/knihy/uid_es-002/ebook.html?p=bilkoviny>

[2] Gregory A PETSKO, Dagmar RINGE. *Protein Structure and Function*. London: New Science Press Ltd, 2004. ISBN 0-9539181-4-9

[3] Carl Braden, John Tooze. *Introduction to Protein Structure*. Second Edition. New York: Garland Publishing, Inc., 1999. ISBN-10: 0815323050

[4] Philip E. Bourne, Helge Weissig. *Structural Bioinformatics.* New Jersey: John Wiley & Sons, Inc., 2003. ISBN 0-471-20200-2

[5] *VMD User's Guide: Rendering methods*. Dostupné z <http://www.ks.uiuc.edu/Research/vmd/current/ug/node54.html>

[6] Marco Tarini, Paolo Cignoni, and Claudio Montani. *Ambient Occlusion and Edge Cueing to Enhance Real Time Molecular Visualization.* IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS, VOL. 12, NO. 5, SEPTEMBER/OCTOBER 2006

[7] *QuteMol*. Dostupné z <http://qutemol.sourceforge.net/>

[8] *PyMol. View 3D Molecular Structures.* Dostupné z <http://www.pymol.org/view>

[9] *PyMol.* Dostupné z <http://www.pymol.org/pymol>

[10] *Jmol: an open-source Java viewer for chemical structures in 3D* Dostupné z <http://jmol.sourceforge.net/>

[11] *Rasmol*. Dostupné z <http://rasmol.org/>

[12] Unipro UGENE. Dostupné z <http://ugene.unipro.ru/>

[13] *PDB File Format. Protein Data Bank Contents Guide: Atomic Coordinate Entry Format Description.*Version 3.30. wwPDB 2008.

[14] RCSB Protein Data Bank. Dostupné z <http://www.rcsb.org/pdb/>

[15] Dave Shreiner, The Kronos OpenGL ARB Working Group. *OpenGL Programing Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1.* Seventh Edition. Addison Wesley 2009. ISBN: 0-321-55262-8

[16] Richard S. Wright, Benjamin Lipchak, Nicholas Haemel. *OpenGL SuperBible: Comprehensive Tutorial and Reference*. Fourth Edition. Addison Wesley, 2007. ISBN: 0-321-49882-8

[17] LibQGLViewer. Dostupné z <http://www.libqglviewer.com/index.html>

[18] Eric Martz and Roger Sayle. *What are the values used by RasMol and Chime for van der Waals radii in the spacefill rendering? How do RasMol and Chime determine which atoms in a PDB file are covalently bonded?.* August 1999; revised and expanded June 25, 2000. Dostupné z <https://www.umass.edu/microbio/rasmol/rasbonds.htm>

[19] Herbert J. Bernstein. *Manual RasMol 2.7.5: CPK Colours.* Updated 17 April 2009. Dostupné z <http://openrasmol.org/doc/rasmol.html#cpkcolours>

[20] Herbert J. Bernstein. *Manual RasMol 2.7.5: Amino Colours*. Updated 17 April 2009. Dostupné z <http://openrasmol.org/doc/rasmol.html#aminocolours>

[21] *Jmol colors.* Dostupné z <jmol.sourceforge.net/jscolors/>

[22] *Real depth in OpenGL / GLSL.* 12 Nov 2010. Dostupné z <web.archive.org/web/20130416194336/http://olivers.posterous.com/linear-depth-in-glsl-for-real>

[23] Song Ho Ahn. *OpenGL Transformation.* 2008-2013. Dostupné z <www.songho.ca/opengl/gl_transform.html>

[24] *The Depth Buffer*. Dostupné z <www.opengl.org/archives/resources/faq/technical/depthbuffer.htm>

[25] Jiří Žára, Bedřich Beneš, Jiří Sochor, Petr Felkel. *Moderní počítačová grafika.* 2. přepracované a rozšířené vydání. Brno: Computer Press, a.s., 2010. ISBN 80-251-0454-0

[26] Michael Krone and Katrin Bidmon and Thomas Ertl. *GPU-based Visualisation of Protein Secondary Structure.* EG UK Theory and Practice of Computer Graphics (2008), pp. 1–8

[27] *An Efficient Way to Draw Approximate Circles in OpenGL.* SiegeLord's Abode 2006-2013 Dostupné z <http://slabode.exofire.net/circle_draw.shtml>

[28] OpenMP. Dostupné z <http://openmp.org/wp/>

[29] Cosine metrics. Dostupné z <http://www.gettingcirrius.com/2010/12/calculating-similarity-part-1-cosine.html>

[30] Render Moneky. Dostupné z <http://developer.amd.com/tools-and-sdks/archive/legacy-cpu-gpu-tools/rendermonkey-toolsuite/>

[31] MSI Afterburner. Dostupné z <http://event.msi.com/vga/afterburner/index.htm>

[32] Qt Project. Dostupné z <http://qt-project.org/>

[33] The C++ Resources Network, 2013. Dostupné z <http://www.cplusplus.com/>

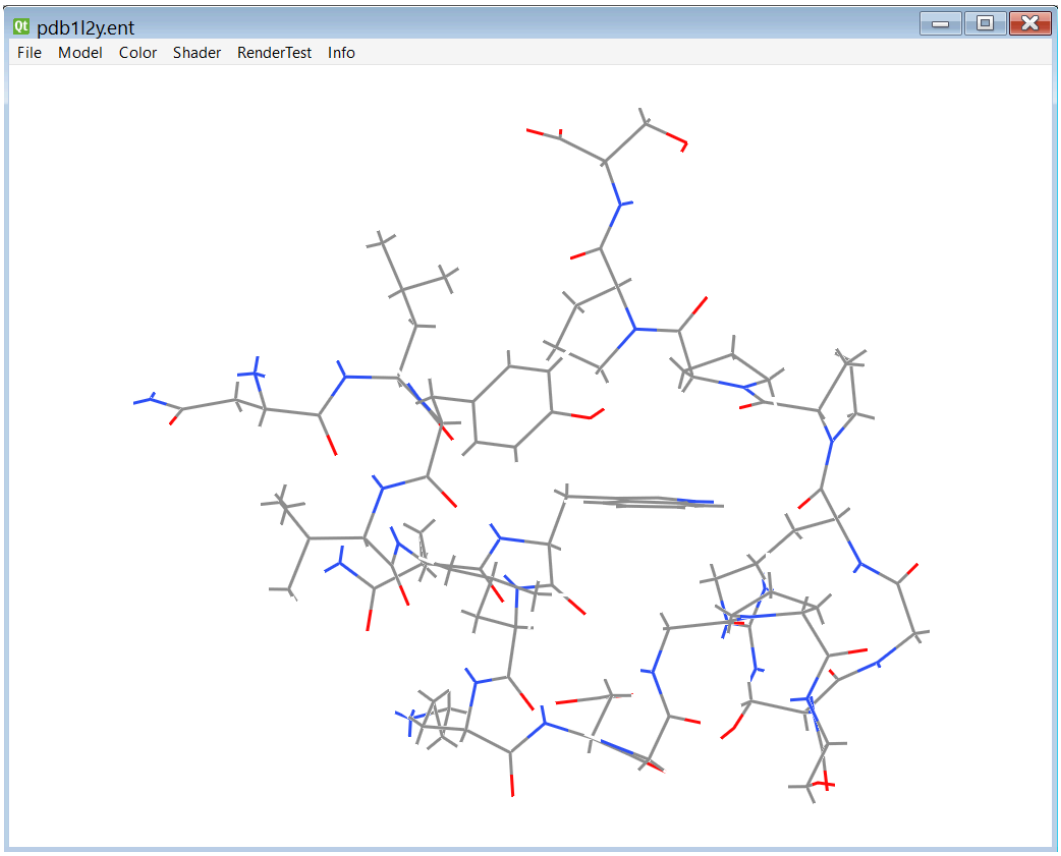
[34] Qt Project Documentation. Dostupné z <http://qt-project.org/doc/>

[35] OpenGL 2.1 Reference Pages. Dostupné z <http://www.opengl.org/sdk/docs/man2/>

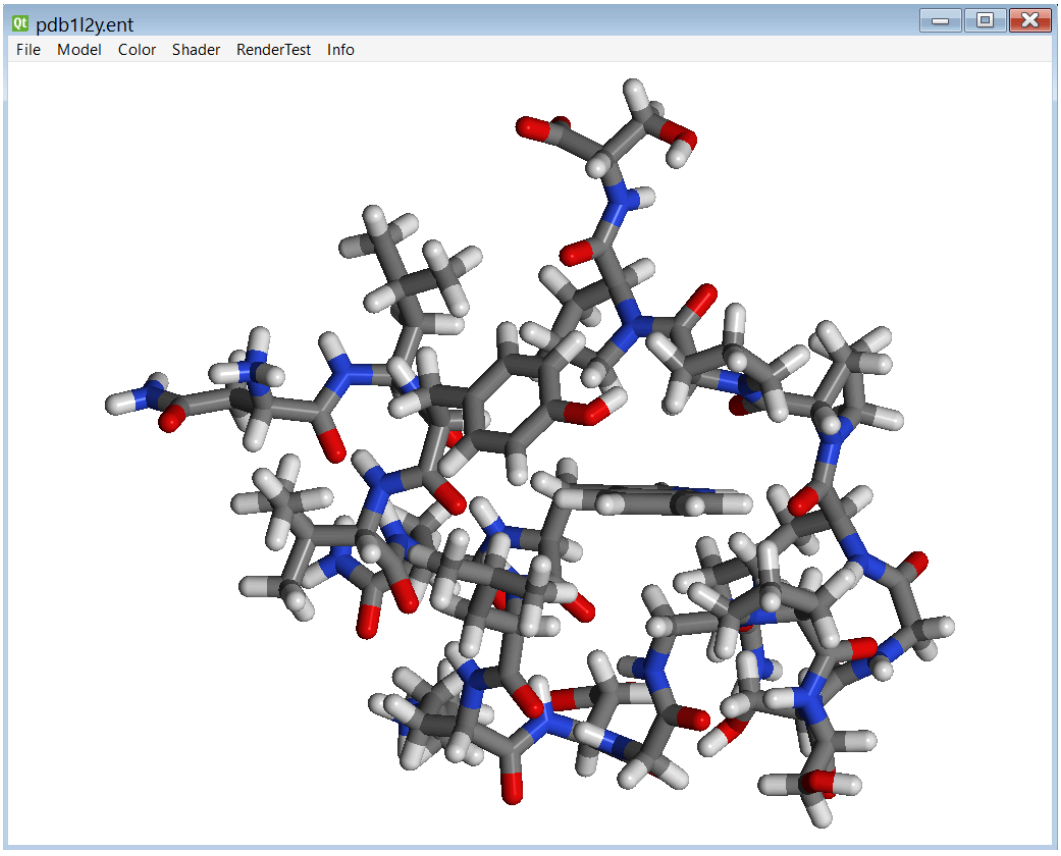
[36] John Kessenich. *The OpenGL Shading Language. Language Version: 1.40.* The Khronos Group Inc. 2009

Seznam příloh

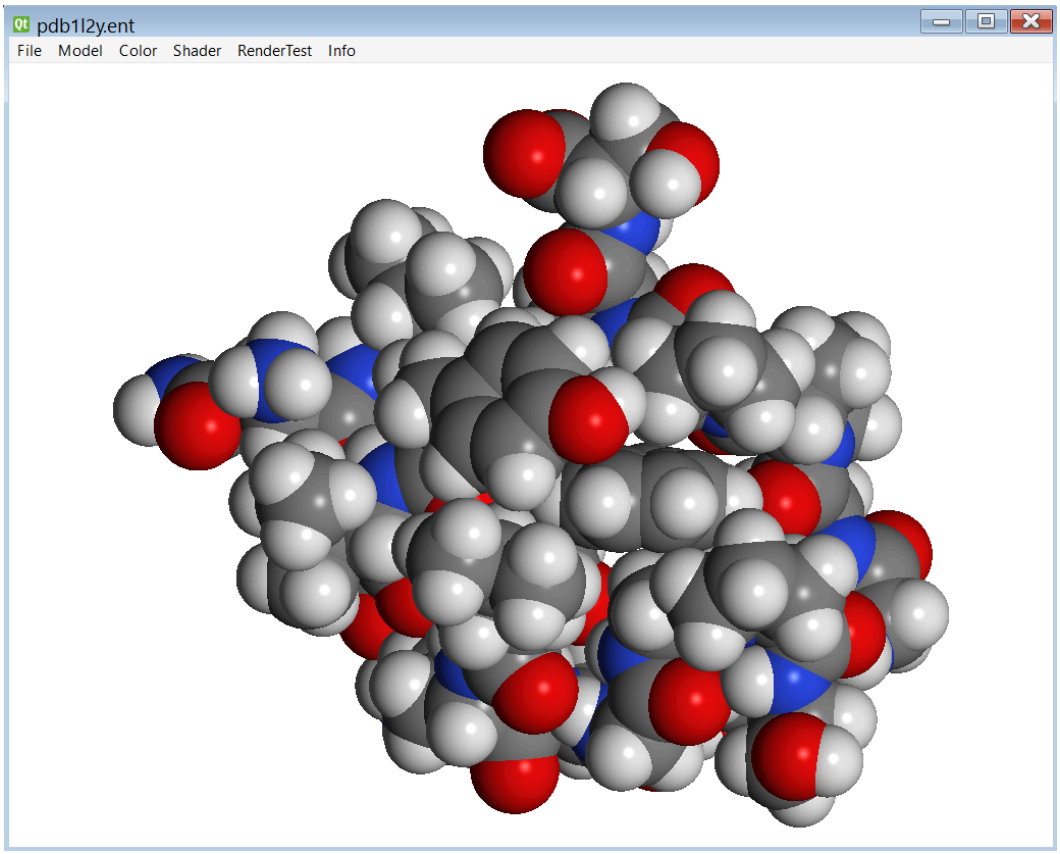
Obrázek I: Vykreslení v režimu wireframe 58
Obrázek II: Vykreslení v režimu sticks 58
Obrázek III: Vykreslení v režimu spacefill 59
Obrázek IV: Vykreslení v režimu balls and sticks 59
Obrázek V: Vykreslení v režimu backbone 60
Obrázek VI: Vykreslení v režimu trace 60
Obrázek VII: Vykreslení v režimu tube 61
Obrázek VIII: Vykreslení v režimu cartoon 61
Obrázek IX: Aplikace s načtenými dvěma molekulami proteinů 62
Obrázek X: Zvýraznění vybrané části 62



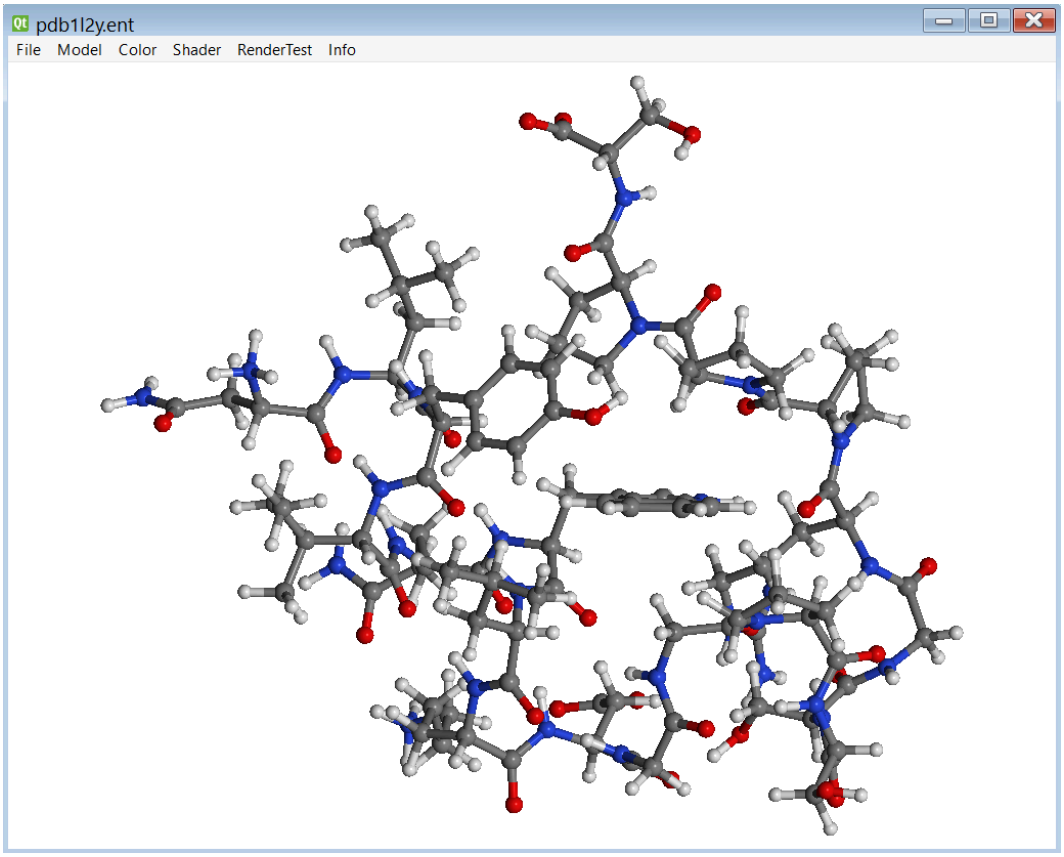
Obrázek I: Vykreslení v režimu wireframe



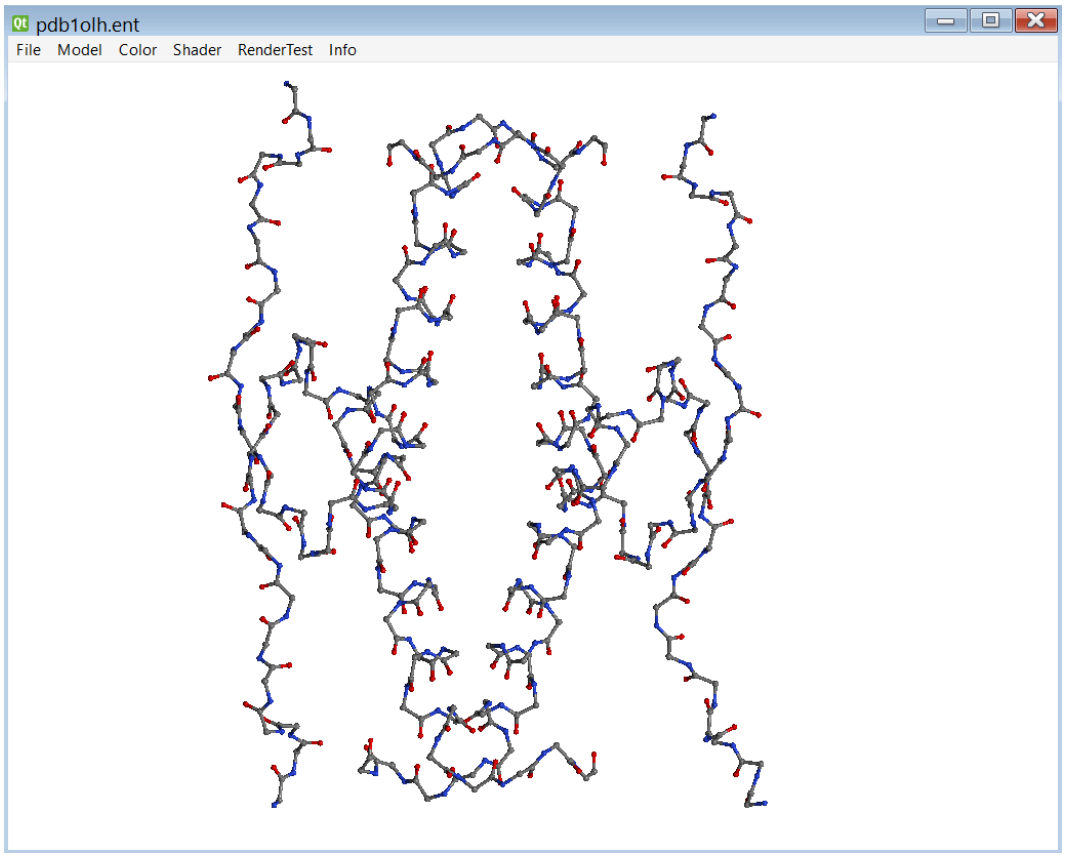
Obrázek II: Vykreslení v režimu sticks



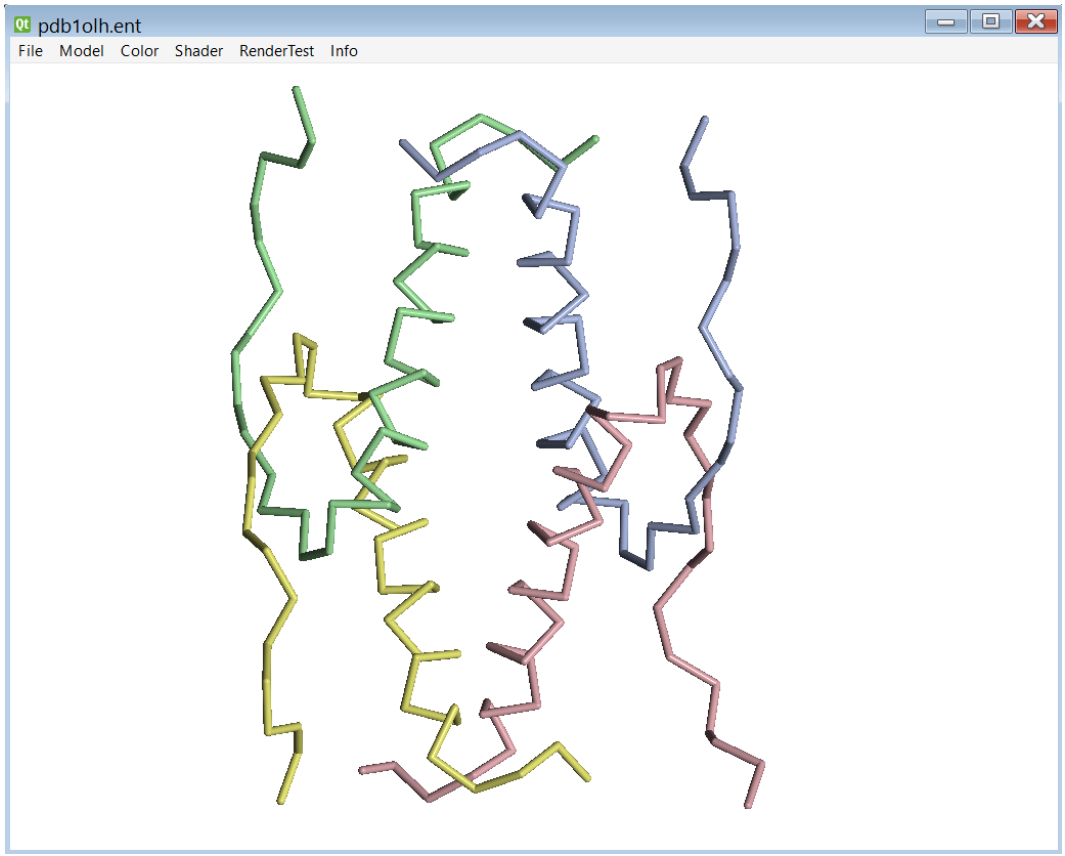
Obrázek III: Vykreslení v režimu spacefill



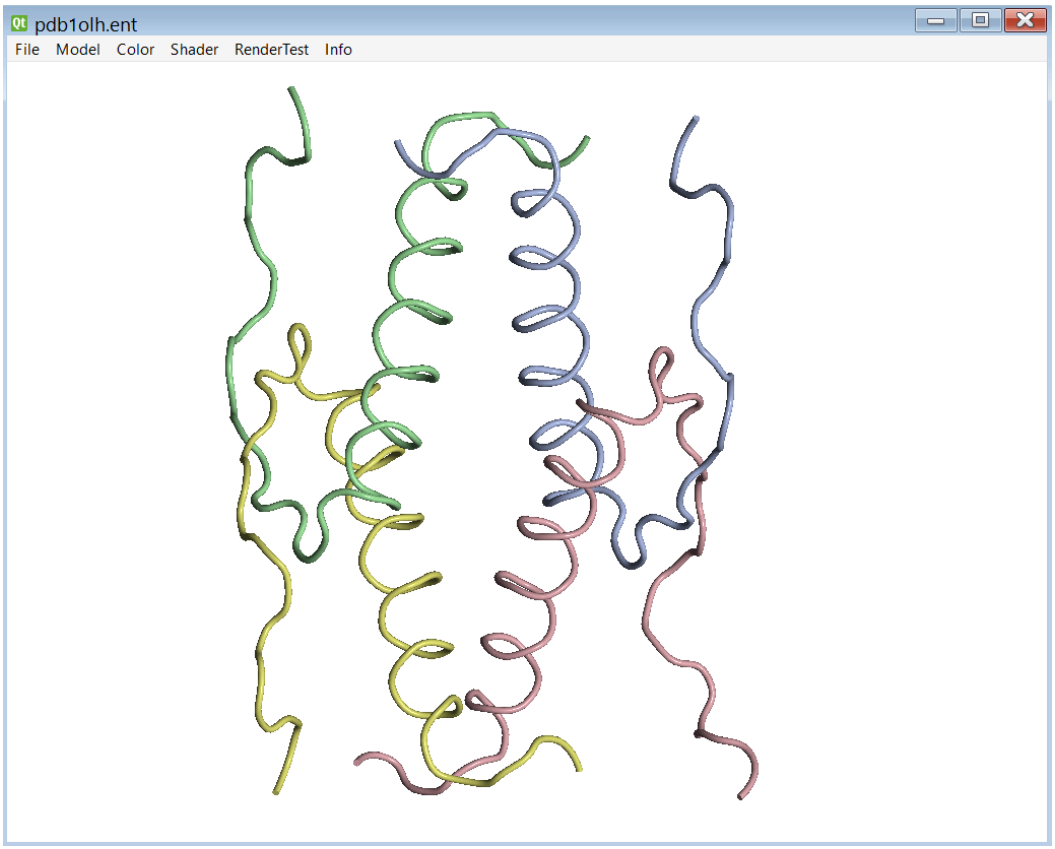
Obrázek IV: Vykreslení v režimu balls and sticks



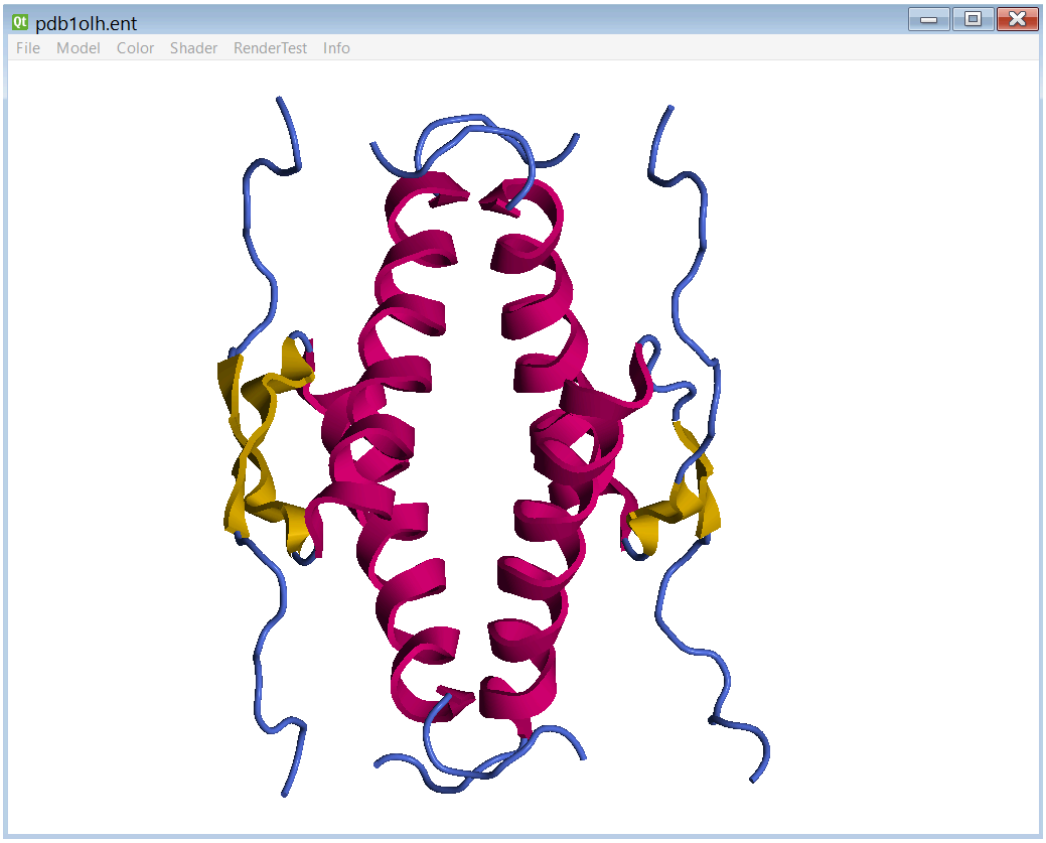
Obrázek V: Vykreslení v režimu backbone



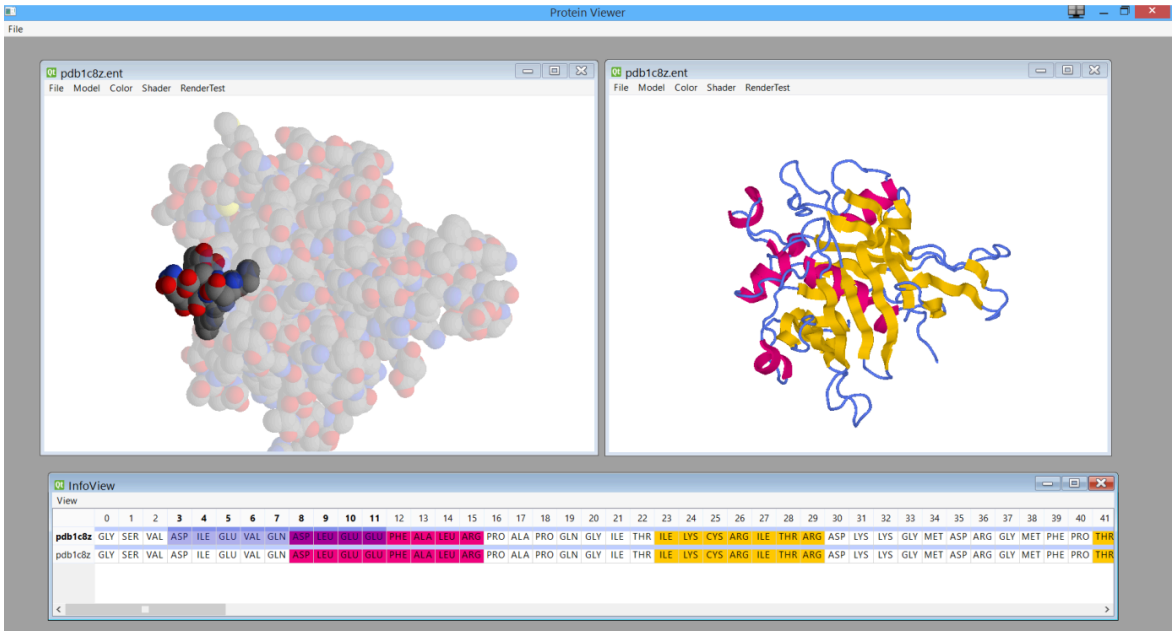
Obrázek VI: Vykreslení v režimu trace



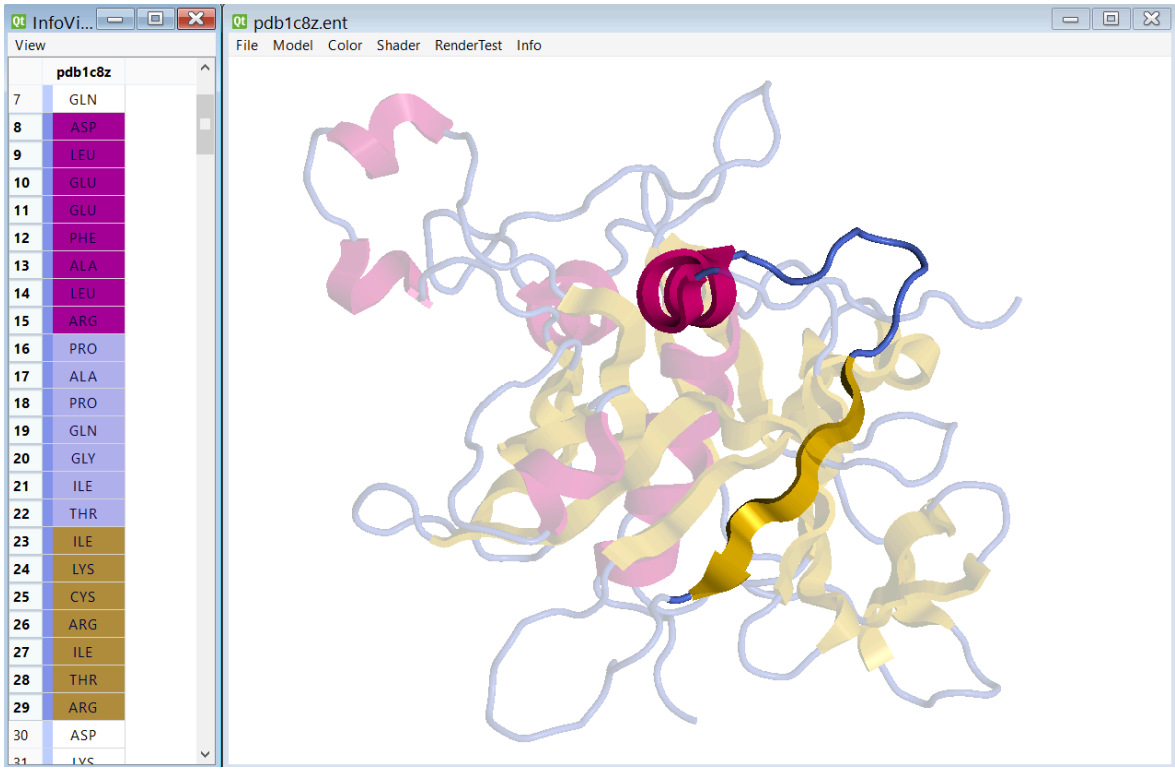
Obrázek VII: Vykreslení v režimu tube



Obrázek VIII: Vykreslení v režimu cartoon



Obrázek IX: Aplikace s načtenými dvěma molekulami proteinů



Obrázek X: Zvýraznění vybrané části